Continuous Detour Queries in Spatial Networks

Sarana Nutanong, Egemen Tanin, Jie Shao, Rui Zhang, and Kotagiri Ramamohanarao

Abstract—We study the problem of finding the shortest route between two locations that includes a stopover of a given type. An example scenario of this problem is given as follows: *"On the way to Bob's place, Alice searches for a nearby take-away Italian restaurant to buy a pizza."* Assuming that Alice is interested in minimizing the total trip distance, this scenario can be modeled as a query where the current Alice's location (start) and Bob's place (destination) function as query points. Based on these two query points, we find the *minimum detour object (MDO)*, i.e., a stopover that minimizes the sum of the distances: 1) from the start to the stopover, and 2) from the stopover to the destination. In a realistic location-based application environment, a user can be indecisive about committing to a particular detour option. The user may wish to browse multiple (k) MDOs before making a decision. Furthermore, when a user moves, the kMDO results at one location may become obsolete. We propose a method for *continuous detour query (CDQ)* processing based on incremental construction of a *shortest path tree*. We conducted experimental studies to compare the performance of our proposed method against two methods derived from existing k-nearest neighbor querying techniques using real road-network data sets. Experimental results show that our proposed method significantly outperforms the two competitive techniques.

Index Terms—Continuous queries, spatial network, spatial databases.

1 Introduction

Location-based services allow users to search nearby facilities and to find the shortest route between two locations. In many cases, a user, who is traveling to a specific destination, may be interested in finding a stopover that does not introduce significant costs to the trip. An application scenario can be given as: *"On the way to Bob's place, Alice searches for a nearby take-away Italian restaurant to buy a pizza."* One approach to addressing the problem in this scenario is: 1) to calculate the shortest path from Alice's location $\boldsymbol{q}_s$ to Bob's place $\boldsymbol{q}_e$; 2) to find the restaurant nearest to that path [4], [27]. In Fig. 1, the shortest path from $\boldsymbol{q}_s$ to $\boldsymbol{q}_e$ is highlighted in gray. Objects $\boldsymbol{a}$, $\boldsymbol{b}$, and $\boldsymbol{c}$ represent restaurants that satisfy the search criteria. The object nearest to the shortest path is $\boldsymbol{b}$, which incurs a deviation of 4 units from the shortest path and an overall trip distance of 21 units. Although this approach is aimed at finding the object residing closest to the route, we argue that it does not necessarily produce the overall shortest path. This is because, this approach uses the deviation from a precomputed shortest path rather than the overall trip distance.

To address this problem, we formulate a new query type, called the *detour query*, which uses the *trip distance* as the optimization measure. Given a set $\mathcal{D}$ of detour objects (stopovers), a starting location $\boldsymbol{q}_s$ and an end location $\boldsymbol{q}_e$, the detour query returns a *minimum detour object (MDO)*. An MDO is an object $\boldsymbol{p}$ in $\mathcal{D}$ that minimizes the TripDist (the sum of: 1) the distance from $\boldsymbol{q}_s$ to $\boldsymbol{p}$, and 2) the distance from $\boldsymbol{p}$ to $\boldsymbol{q}_e$). Fig. 1 illustrates an example of the detour query where the user wishes to travel from $\boldsymbol{q}_s$ to $\boldsymbol{q}_e$, the MDO in this scenario is $\boldsymbol{a}$, which provides the TripDist of 15 units (6 units shorter than the solution from the previous approach).

In order to provide support for realistic location-based applications, we tackle the aforementioned problem from two aspects. The first aspect is the ability to browse and to compare multiple results. By displaying k MDOs at a time, we allow a user to browse and to select the option with which they are most satisfied. In Fig. 1, for example, the 2 MDOs with respect to $\boldsymbol{q}_s$ and $\boldsymbol{q}_e$ are $\langle \boldsymbol{a}, \boldsymbol{b} \rangle$. Although $\boldsymbol{a}$ provides the smallest TripDist, $\boldsymbol{b}$ can be more appealing in terms of food quality or prices.

The second aspect is continuous monitoring of k MDOs. Ongoing kMDO monitoring provides users who wish to take time browsing query results with up-to-date information in the same manner as other continuous spatial queries [3], [4], [28]. A user may browse to gather information without a current intention to commit to a particular decision [1]. As a result, browsing users usually take more time to make decisions than users who are searching for something specific [1], [2], [14], [25]. An application scenario where browsing applies is given as "on the way to Bob's party, Alice is consulting Bob whether she should drop by an Italian restaurant and purchase take away pizzas to bring to the party." Making a detour decision in this scenario can be an ongoing process. It involves Alice and Bob collaboratively browsing possible detour options. Using CDQ, Alice could obtain results and discuss them with Bob. CDQ also allows Alice to inform Bob when results change and ensures that they always have up-to-date information to help make a detour decision whenever they are happy.

A straightforward approach to solving the *continuous detour query (CDQ)* problem is to evaluate the kMDO at each intersection along the trajectory. Specifically, we may

- *S. Nutanong is with the Institute for Advanced Computer Studies, University of Maryland, College Park, MD 20742. E-mail: nutanong@umiacs.umd.edu.*
- *E. Tanin and K. Ramamohanarao are with the NICTA Victoria Research Laboratory, Victoria, Australia and the Department of Computer Science and Software Engineering, University of Melbourne, Victoria 3010, Australia. E-mail: {egemen, rao}@csse.unimelb.edu.au.*
- *J. Shao and R. Zhang are with the Department of Computer Science and Software Engineering, University of Melbourne, Victoria 3010, Australia. E-mail: {jsh, rui}@csse.unimelb.edu.au.*

Manuscript received 15 Sept. 2010; revised 20 Dec. 2010; accepted 16 Jan. 2011; published online 9 Feb. 2011.
Recommended for acceptance by T. Sellis.
For information on obtaining reprints of this article, please send e-mail to: tkde@computer.org, and reference IEEECS Log Number TKDE-2010-09-0499.
Digital Object Identifier no. 10.1109/TKDE.2011.52.

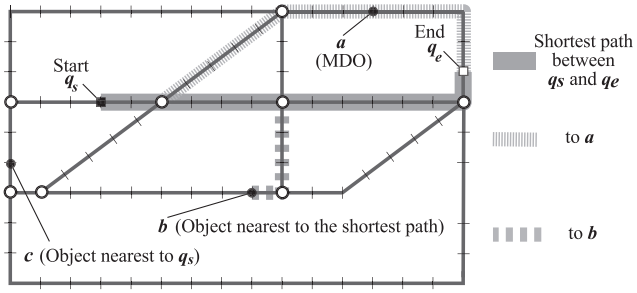


Fig. 1. Minimum detour object with respect to a starting point q_s and an end point q_e.

consider the TRIPDIST, $(\text{DIST}(q_s, p) + \text{DIST}(p, q_e))$, as an aggregate distance function [19] with two query points q_s and q_e. Then, we search for k objects with the minimum TRIPDISTs for the give locations of q_s and q_e. Since the starting location q_s changes over time, this approach incurs repetitive evaluation of network distances. We use this method as one of the comparators in our experiments.

In this paper, we propose a CDQ solution which incrementally evaluates the kMDO results at different intersections according to the TRIPDIST measure. Consequently, repetitive distance evaluation is avoided.

The contributions of this paper are summarized as follows:

- We formalize the problem of continuous evaluation of k MDOs for a moving query and a fixed destination in a spatial network.
- We propose a novel solution to the CDQ problem that: a) is capable of handling multiple MDOs, b) does not require access to all data objects, and c) does not incur repetitive distance evaluation as the query point moves.
- We conduct extensive experiments with a real road network data set and realistic application scenarios. Our experimental results show that our proposed method significantly outperforms a competitive method derived from the *aggregate k nearest neighbor* (AkNN) querying domain [19], [26].

The rest of the paper is organized as follows: Section 2 presents the problem setting. In Section 3, we discuss related work, i.e., nearest neighbor (NN) queries in spatial networks and route planning queries. Section 4 presents background knowledge on the Dijkstra's algorithm and Voronoi diagrams (VD) which will be needed for our explanations. In Section 5, we describe our proposed solution. A competitive solution and discussion are presented in Sections 6 and 7, respectively. In Section 8, we report experimental results. This paper is concluded in Section 9.

2 Problem Formulation

A spatial network $G(N, E)$ is represented by a set N of nodes (intersections) and a set E of edges (road segments). For any given two points p_1 and p_2 on $G(N, E)$, the distance $\text{DIST}(p_1, p_2)$ is the distance via the shortest path from p_1 to p_2. For ease of exposition, we use undirected graphs in our presentation, which means that $G(N, E)$ satisfies *symmetry* and *triangle inequality* among the other metric space conditions.[1] Notations frequently used in this paper are summarized in Table 1.

TABLE 1
Frequently Used Notations

Notation	Meaning
$G(N, E)$	Spatial network.
$\text{EDGE}(n_i, n_j)$	Edge that connects n_i to n_j.
$\mathcal{D}$	Set of data objects represented as points.
q	Moving query point.
q_s	Start location.
q_e	End location.
$\text{PATH}(p_1, p_2)$	Shortest path from p_1 to p_2.
$\text{DIST}(p_1, p_2)$	Length of $\text{PATH}(p_1, p_2)$.
$\text{TRIPDIST}(q_s, p, q_e)$	$\text{DIST}(q_s, p) + \text{DIST}(p, q_e)$.
MDO	Minimum detour object.
k	Number of resultant objects requested.
kMDO	List of k MDOs ranked using TRIPDIST.

We first define the detour query as a query that returns the MDO, and also give the definition of MDO as follows:

Definition 1 (Minimum Detour Object). *Given a set $\mathcal{D}$ of data objects in a spatial network $G(N, E)$, the MDO is the object p in $\mathcal{D}$ with the smallest $\text{TRIPDIST}(q_s, p, q_e)$, where q_s and q_e represent the coordinates of start and end locations, respectively.*

Based on this MDO definition, the kMDO is defined as a sorted list of k detour objects with the smallest TRIPDISTs.

Second, our proposed query is defined as follows:

Definition 2 (Continuous Detour Query). *Given a set $\mathcal{D}$ of data objects, a moving query point q and a destination q_e in $G(N, E)$, the CDQ query continuously finds the kMDO with respect to a q and q_e.*

3 Related Work

3.1 NN Queries in Spatial Networks

Papadias et al. [20] introduced two frameworks for the spatial-network kNN query: the *incremental euclidean restriction (IER)* and *incremental network expansion (INE)*. IER applies the property that the euclidean distance between any two network nodes is a lower bound of their network distance to prune the search space. INE performs network expansion similar to the Dijkstra's algorithm [9] from the query point and examines data objects in the order they are encountered. They showed that INE performs better than IER in general.

As an optimization of IER, Deng et al. [8] proposed incremental *lower bound constraint (LBC)*. The LBC method calculates distance lower bounds of objects for pruning purposes. Hence, the workload from network distance calculation is greatly reduced.

Kolahdouzan and Shahabi [10] presented a *Voronoi-based network nearest neighbor* (VN^3) approach to evaluate the kNN query by decomposing the data space using the first-order network Voronoi diagram with respect to data objects.

1. In the formulation of our algorithms, however, the distance function DIST(,), as well as, the notations EDGE(,) and PATH(,) are considered directional. Our proposed method is therefore applicable to directed graphs, where the symmetry condition does not hold.

TABLE 2
Queries and Techniques Related to the Continuous Detour Query

Technique	Optimization Goal	Temporality	Number of Results	Global Access
TPQ [13]	*Trip Distance*	Snapshot	*Multiple*	*No*
OSR [23]	*Trip Distance*	Snapshot	*Multiple*	*No*
OSR with AWVD [24]	*Trip Distance*	*Continuous*	Single	Yes
IRNN [27]	Deviation	Snapshot	*Multiple*	*No*
PNN Monitoring [4]	Deviation	*Continuous*	*Multiple*	*No*
Road Network AkNN [26]	*Trip Distance*	Snapshot	*Multiple*	*No*
Road Network Group kNN [21]	*Trip Distance*	Snapshot	*Multiple*	Yes
IkSPT (Proposed)	*Trip Distance*	*Continuous*	*Multiple*	*No*

Finding the k NNs of a query point $\boldsymbol{q}$ is done by: 1) identifying the first NN using the Voronoi diagram; 2) deriving the subsequent NNs from neighboring Voronoi cells. Other precomputation-based techniques include kNN algorithms [7], [22] that use precomputed shortest path information stored in quadtrees and grid-based data structures.

Next we discuss continuous NN (CNN) problems in spatial networks. In this paper, we consider the setting of moving query objects and stationary data objects. We omit discussion on another class of CNN techniques [7], [15] which address problems in the setting of moving data objects.

Kolahdouzan and Shahabi [11] proposed the *upper bound algorithm (UBA)* for continuous kNN queries in a spatial network. The algorithm retrieves $(k+1)$ NNs with respect to a given location and calculates an upper bound. This upper bound is used to eliminate kNN computations between locations that kNN does not change.

Cho and Chung [5] proposed a continuous kNN technique that performs snapshot kNN queries at the intersections on the query path. They showed that kNN results between any two intersections can be inferred from those of the intersections. They also formulated an algorithm to find points where kNN changes for a predetermined query trajectory.

Nutanong et al. [16] proposed a technique called the V*-diagram. In addition to the regular k NNs with respect to a given query point, their technique retrieves x auxiliary objects. The value of x is generally in the same order of magnitude as k. Retrieval of these auxiliary objects provides additional search scope to allow the query point to move while retaining enough information to continuously produce kNN results. As a result, the access cost is reduced.

These NN techniques are aimed at finding an object with the smallest distance with respect to a single query point. Using the NN query to solve the detour query problem (by assigning the starting location to the query point) may result in an impractical route, especially when the nearest object is in the opposite direction to the destination. For example, in Fig. 1, the nearest restaurant with respect to $\boldsymbol{q}_s$ is $\boldsymbol{c}$, which provides the TRIPDIST greater than those of $\boldsymbol{a}$ and $\boldsymbol{b}$.

3.2 Route Planning Queries

Our kMDO monitoring problem can be categorized as a route planning problem. Specifically, we use the term *route planning* to refer to problems of finding a route through multiple destinations with respect to given routing requirements. Queries and techniques closely related to our problem are summarized in Table 2, where each is categorized by:

1. **optimization goal:** the distance measure it aims to minimize;
2. **temporality:** whether it produces snapshot or continuous results;
3. **number of results:** the number of resultant objects produced/monitored;
4. **global access:** whether it incurs access to all objects and nodes in the data set.

For example, to produce a kMDO result for one location, a global-access method examines all data objects and network nodes in the entire network.

The last row displays our proposed technique, *incremental order-k shortest path tree (IkSPT)*. To illustrate how our proposed technique fits into the existing literature, the table also shows the differences between existing techniques and ours. For example, TPQ [13] and OSR [23] are snapshot queries, while the continuous OSR monitoring method [24] can handle a single result at a time and requires access to all objects to construct a Voronoi diagram. The PNN monitoring technique [4] aims to minimize the deviation from a dynamically changing path instead of minimizing the TRIPDIST. In summary, none of these existing techniques can monitor multiple CDQ results in a spatial network. Detailed discussions of these queries and techniques are given as follows:

Li et al. [13] proposed the *trip planning query (TPQ)*. Given a list L of types of objects, the TPQ finds the shortest route that includes objects of those types in L with respect to given start and end points. For example, a user can be interested in visiting a post office and a gas station before going to work. The TPQ finds the shortest route to work that includes a post office and a gas station. A similar query, called the *optimal sequenced route (OSR) query*, was proposed by Sharifzadeh et al. [23]. Given a point $\boldsymbol{q}$ and a sequence S of object types, the OSR query finds the route that starts at $\boldsymbol{q}$ that orderly passes through object types in S, and minimizes the traveling distance. The main difference between OSR and TPQ is that OSR is ordered, while TPQ is not.

Sharifzadeh and Shahabi [24] present a safe region-based solution to the OSR query using the *additively weighted Voronoi diagrams (AWVD)*. In an AWVD, each generator point $\boldsymbol{p}_i$ is the location of the first visited object of each possible route R_i and the associated weight w_i is calculated from the traveling distance between the first and the last objects. The boundary between the AWVD cells of two routes $R_1 : (\boldsymbol{p}_1, w_1)$ and $R_2 : (\boldsymbol{p}_2, w_2)$ is a hyperbolic curve,

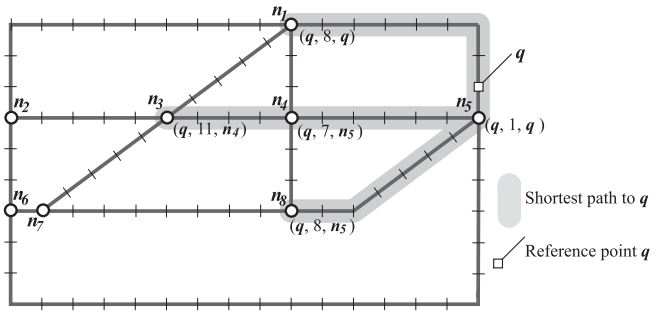


Fig. 2. End product of the Dijkstra's algorithm where each node contains a label (q, d, n_h) denoting the reference point q, the distance d to q, and the node n_h from which the distance d is derived, respectively.

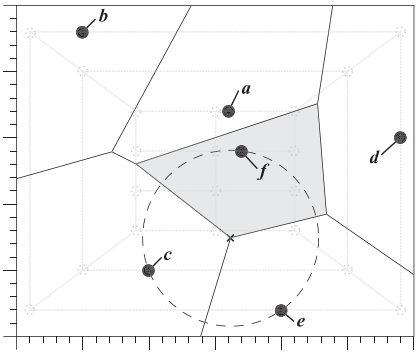


Fig. 3. Voronoi diagram in euclidean space.

$\|v - p_1\| + w_1 = \|v - p_2\| + w_2$. That is, R_1 and R_2 provides the same total traveling distance with respect to the starting location. Since associated weights w_i can be any positive value, this technique can also be applied to TPQ.

Yoo and Shekhar [27] proposed the *in-route nearest neighbor (IRNN)* query. The IRNN query finds a stopover that minimizes a deviation from a given path. Since a user can deviate from the path via only a node, this becomes a closest pair problem [6] between two sets of locations: the set of possible stopovers, and the set of nodes along the preferred path.

Chen et al. [4] studied the problem of *path nearest neighbor (PNN)* for a moving query point q. Their studies include algorithms to maintain the shortest path from q to a fixed destination; and to monitor k PNNs, i.e., the k objects with the minimum deviations from the shortest path.

Based on the INE and IER frameworks [20], Yiu et al. [26] proposed a spatial-network variant of the *aggregate kNN (AkNN) query* [19]. The AkNN query retrieves k objects with the smallest aggregate distances to a set of query points. Given a start q_s and destination q_e in an undirected graph, the aggregate distance is equivalent to TRIPDIST when the aggregate function is SUM and the query set is $\{q_s, q_e\}$. We adopt the principle of AkNN querying and the INE framework to formulate a comparator (in Section 6).

Safar [21] proposed an algorithm to find *group k nearest neighbors* in a spatial network. That is, a set of k objects that minimize the sum distances from a given query set. Similar to the VN^3 approach [10], the algorithm utilizes the network Voronoi diagram and precomputed distances.

4 Preliminaries

4.1 Network Distance Calculation

This section provides background understanding of network distance calculations. Given two points x and y in a network/graph, Dijkstra [9] introduced an algorithm to calculate the shortest path from x to y based on the concept of best-first graph traversal. Specifically, to find the shortest path PATH(x, y), the algorithm uses y as a reference point and incrementally examines surrounding nodes until x is covered, or vice versa. This algorithm is also known as "distance scan" since all nodes with distances smaller than that of x have to be visited. In addition, one may resume the computation from where it is terminated to obtain the shortest path from y to any node farther than x.

In our problem setting, since we have to compute the distances from multiple detour objects to a single destination q_e, we further explain the Dijkstra's algorithm using Fig. 2. To calculate PATH(n_3, q), the algorithm incrementally discovers surrounding nodes according to their distances to q. In this example, n_5 is discovered first with a distance of 1 unit. Next, n_4, n_1, and n_8 are discovered with distances of 7 (via n_5), 8 (via q), and 8 units (via n_5), respectively. The search halts when n_3 is discovered.

Fig. 2 also shows that each of the nodes involved in the search is associated with a label (q, d, n_h) where q denotes the reference point, d denotes the shortest distance, and n_h denotes the node from which the shortest distance is derived. The shortest path from n_3 to q can be obtained by recursively traversing the next hop n_h in the label (q, d, n_h) until q is reached. In this case, we obtain $\langle n_3, n_4, n_5, q\rangle$ as the shortest path. As displayed in Fig. 2, a *shortest path tree* (SPT) (where the reference point q is the root) is formed by linking the next hop n_h of all nodes involved in the calculation.

4.2 Voronoi Diagrams

This section provides discussion on variants of the Voronoi diagram as background knowledge for our proposed method. The most basic form of Voronoi diagrams [17] is the Voronoi diagram in euclidean space with the order value of 1. Fig. 3 shows the VD of six data objects as generators in euclidean space (the underlying network connectivity is ignored). Each generator is associated with a cell in which the generator dominates. For example, the Voronoi cell VC(f) is a region containing points v such that the distance from v to f is not greater than the distance from v to any other generator. The boundary between two Voronoi cells p_i and p_j is defined by the bisector BT(p_i, p_j), i.e., a set of locations equidistant to p_i and p_j. For example, the boundary of VC(f) consists of the bisectors BT(f, a), BT(f, c), BT(f, d), and BT(f, e). This technique allows the NN problem to be treated in the same manner as a point-location problem. That is, locating the Voronoi cell in which the query point belongs.

Fig. 4 displays a technique to construct a *network*[2] *VD*, where three boundaries (bisectors) of a sample cell VC(f)

2. The qualifier "network" is omitted when context is clear.

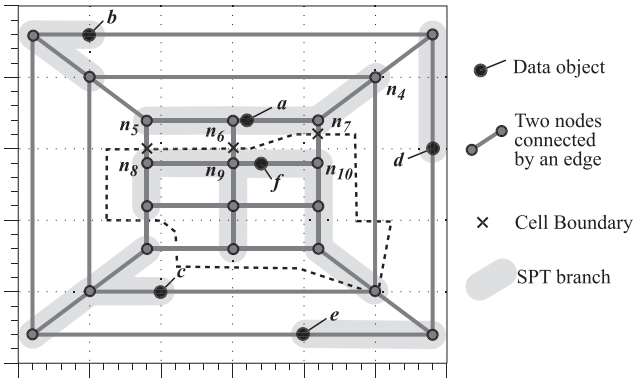


Fig. 4. Shortest path trees and a Voronoi cell.

are marked as crosses. A network VD can be constructed using the *shortest path tree* technique [18]. The technique is based on the best-first search principles similar to that of the Dijkstra's algorithm [9]. Specifically, by replacing a single reference point with the objects (generators), we can use the Dijkstra's algorithm explained in Section 4.1 to compute SPTs of multiple objects. An end product is a set of SPTs where each is associated with its nearest generator. Boundaries of Voronoi cells are obtained using the distance information embedded in each node to find points equidistant to two generators.

Fig. 4 shows a forest of shortest path trees constructed based on the generators $\{a, \ldots, f\}$. We consider this forest as a single tree when each branch is associated with its nearest generator and each generator is tied to a root. For example, the branch of generator a has $\{n_4, \ldots, n_7\}$. The boundary between $\mathrm{VC}(a)$ and $\mathrm{VC}(f)$ on $\mathrm{EDGE}(n_5, n_8)$ is calculated based on $\mathrm{DIST}(n_5, a)$ and $\mathrm{DIST}(n_8, f)$, which are 7 and 8, respectively. Therefore, the point on $\mathrm{EDGE}(n_5,n_8)$ that makes a equidistant to f is $\mathrm{PNT}(n_5,n_8,2)$, the point 2 units into $\mathrm{EDGE}(n_5,n_8)$ (measured from n_5). This point is shown as the cross on $\mathrm{EDGE}(n_5, n_8)$. A Voronoi cell can be obtained by applying this calculation to edges adjacent to its generator and the surrounding generators. For example, $\mathrm{VC}(f)$ (the region enclosed by dotted lines) is bounded by the bisectors between f and its three neighboring generators a, c, and e. This technique can be used to provide answers for NN queries in a spatial network. For example, given a VD and an SPT for restaurants in a road network, the nearest restaurant can be obtained by locating the Voronoi cell that contains the user's location, while the SPT provides the shortest route to that restaurant.

In the previous example, we assume that the preference for a particular restaurant is decided by the distance solely. The *additively-weighted VD (AWVD)* [18] is devised to address NN problems with an offset measure. For example, restaurants can be offset by their food prices to increase the desirability of cheaper restaurants. In the CDQ problem, each data object is associated with an offset derived from the distance to one common destination. However, the AWVD construction technique [18] is inapplicable to the CDQ problem because it only handles a single resultant object, and requires global access to network nodes and data objects. This technique forms the basis to describe our proposed method in the next section.

Fig. 5 provides an example of the SPT technique using the additive weight concept. Fig. 5a presents the order in

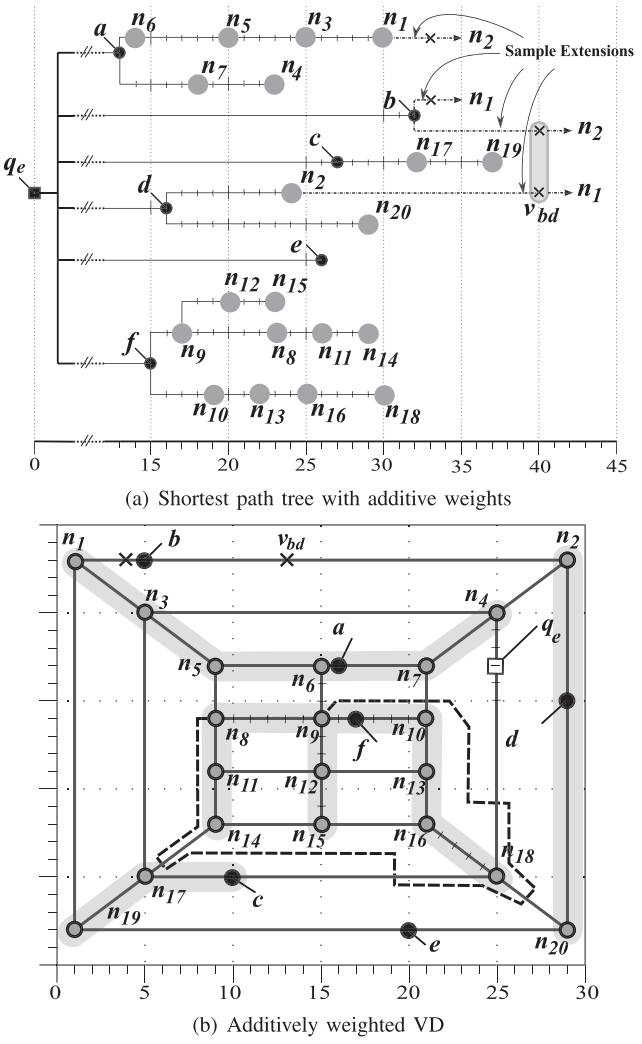


Fig. 5. Construction of an AWVD using the SPT technique.

which the nodes are included in the SPT. The x-axis represents the TRIPDIST measure. The SPT has six branches, where each branch corresponds to each of the six generators a to f. The initial position along the x-axis of an SPT branch corresponds to the additive weight (the distance from q_e) of the corresponding generator. For example, the additive weight of a is 13 units so its initial position is 13. The first node to be included in SPT is n_6 providing the TRIPDIST of 14. The SPT incrementally expands according to the additively weighted distance to the generators. The process stops when all nodes are labeled. In this example, the last node labeled is n_{19}.

After the SPT is constructed, the boundaries between Voronoi cells are calculated by finding edges $\mathrm{EDGE}(n_i, n_j)$ where n_i and n_j belong to two different generators. In Fig. 5b, the boundary between $\mathrm{VC}(b)$ and $\mathrm{VC}(d)$ on $\mathrm{EDGE}(n_1, n_2)$ is the point location v_{bd} such that $\mathrm{TRIPDIST}(v_{bd}, b, q_e)$ is equal to $\mathrm{TRIPDIST}(v_{bd}, d, q_e)$. In this case, v_{bd} is $\mathrm{PNT}(n_1, n_2,12)$. This boundary is also shown as the extension to n_2 on the branch of b and the extension to n_1 on the branch of d in Fig. 5a.

5 Proposed Solution

In this section, we present our proposed solution, *incremental order k shortest path tree* in the following steps.

First, we generalize the SPT technique to the order-k SPT (kSPT). Second, we show how kSPT construction can be done in an incremental manner. Finally, we present a method to compute the kMDO of any point in the network from a kSPT.

5.1 Order-k Shortest Path Tree

In this section, we present a method to construct a kSPT by introducing overlaps between SPT 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 define the network data structure used by our proposed method. A network/graph is represented using the adjacency-list format, i.e., a list of nodes where each node entry contains information regarding its adjacent nodes. A data structure "Node" is defined as follows:

Definition 3 (Node structure). *The structure of a node $\boldsymbol{n_i}$ contains the following attributes:*

- ID: *the node identification.*
- Adjacency list (AdjList): *a list of edges to/from immediate neighbors and associated weights. (For a directed graph, an adjacency list may comprise three edge types: incoming, outgoing, and bidirectional.)*
- Label list (LabelList): *a list of (at most) k labels. For each label $(\boldsymbol{p}, d, \boldsymbol{n_h})$ in the label list of $\boldsymbol{n_i}$,*
  - *$\boldsymbol{p}$ represents a detour object,*
  - *d represents TRIPDIST$(\boldsymbol{n_i}, \boldsymbol{p}, \boldsymbol{q_e})$, and*
  - *$\boldsymbol{n_h}$ represents the next hop in order to get to $\boldsymbol{p}$.*
- Type: *a node type is "Labelable" by default and becomes "Permanent" upon completion of k labels.*

Fig. 6 shows a 2SPT of the network in Fig. 5b. The figure also shows the order in which nodes are processed according to the TRIPDIST measure. Each node in the 2SPT consists of two labels: *first* and *second*. For example, $\boldsymbol{n_6}$ appears twice in the 2SPT in the branches of generators $\boldsymbol{a}$ and $\boldsymbol{f}$, respectively. The first label of $\boldsymbol{n_6}$ corresponds to the generator $\boldsymbol{a}$ and TRIPDIST$(\boldsymbol{n_6}, \boldsymbol{a}, \boldsymbol{q_e})$ of 14 units. The second label of $\boldsymbol{n_6}$ corresponds to $\boldsymbol{f}$ and the TRIPDIST of 20 units. Note that the portion containing the first labels (highlighted in gray) is identical to the order-1 SPT shown in Fig. 5a.

We describe kSPT construction steps in Algorithm 1. The algorithm accepts a set $\mathcal{D}$ of objects, a value of k, a graph $G(N, E)$, and a destination $\boldsymbol{q_e}$ as input. The output kSPT is provided as $G(N, E)$ with kMDO information embedded. Specifically, we introduce k labels for each $\boldsymbol{n_i}$ in N. The initialization (Lines 1 to 10) includes the following steps.

- First, a priority queue PQ is initialized. A priority-queue entry is a tuple $(\boldsymbol{n}, \boldsymbol{p}, d, \boldsymbol{n_h})$, where $\boldsymbol{n}$ is the node to which the entry corresponds, and the other three elements $\boldsymbol{p}$, d, and $\boldsymbol{n}_h$ form a labeling candidate for an entry in $\boldsymbol{n}$.LabelList. Entries in PQ are ranked according to the labeling distance d.
- Second, for each object $\boldsymbol{p}$, we create a node entry $\boldsymbol{n}_p$ and insert it into $G(N, E)$ where affected edges in E are accordingly modified (Lines 3 and 4). We create a priority queue entry for $\boldsymbol{n_p}$ with the associated detour object $\boldsymbol{p}$ and the labeling distance d of DIST$(\boldsymbol{n}_p, \boldsymbol{q_e})$. The next hop $\boldsymbol{n_h}$ (shown as a dash "-" sign) is inapplicable in this case since $\boldsymbol{n_p}$ is already at $\boldsymbol{p}$. The entry $(\boldsymbol{n_p}, \boldsymbol{p}, d, -)$ is inserted in to PQ (Line 7).
- Third, for each network node, we create an empty label list and set the node type to "Labelable" (Lines 8 to 10).

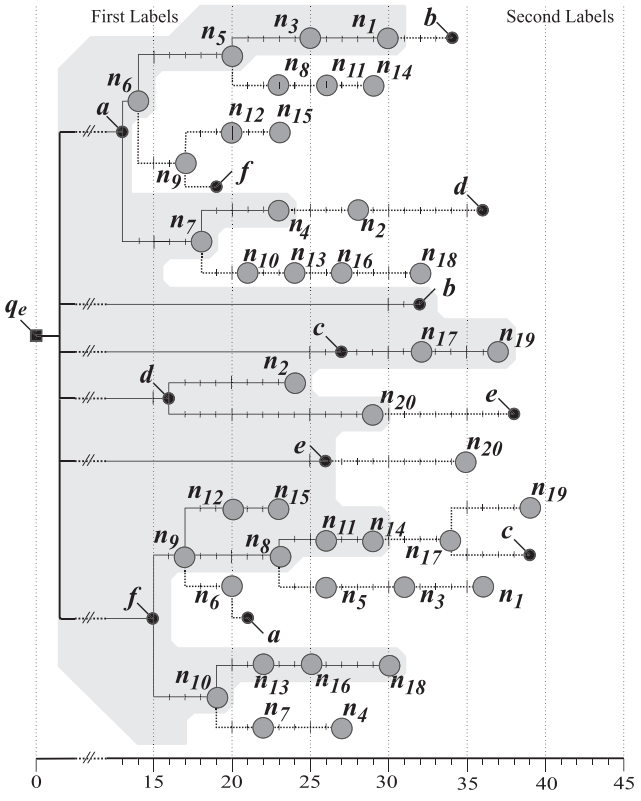


Fig. 6. Order-k SPT with k of 2.

Best-first search is handled by the *while loop* (Lines 11 to 20). The first step of each iteration is to dequeue the head entry $(\boldsymbol{n}, \boldsymbol{p}, d, \boldsymbol{n_h})$ from PQ (Line 12). Since we are interested in only the first k labels of each node, the entry is ignored if the node's label list already contains k labels, i.e., the node is "Permanent." In addition, to ensure that each node is associated with k unique MDOs, the entry is also ignored if there exists an entry with Object $\boldsymbol{p}$ as the associated detour object in the label list. Otherwise, a label $(\boldsymbol{p}, d, \boldsymbol{n_h})$ is added to the node's label list (Line 14). The node type becomes permanent if this label is the kth entry in the label list (Lines 15 and 16). In Lines 17 to 20, for each node $\boldsymbol{n_a}$ such that there exists an edge that connects $\boldsymbol{n}_a$ to $\boldsymbol{n}$, we create a PQ entry e_a, $(\boldsymbol{n_a}, \boldsymbol{p}, d + \omega, \boldsymbol{n})$, where

- $\boldsymbol{n_a}$ is the node to which this entry corresponds;
- $\boldsymbol{p}$ is the associated detour object;
- $(d + \omega)$ is the labeling distance calculated by adding the current labeling distance d to the weight of EDGE$(\boldsymbol{n_a}, \boldsymbol{n})$;
- $\boldsymbol{n}$ is the node from which the labeling distance is derived.

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

TABLE 3
First and Second Labels of Network Nodes
(with Some Entries Omitted)

Node	First Label	Second Label
a	$(a, 13, -)$	$(f, 21, n_9)$
b	$(b, 32, -)$	$(a, 34, n_1)$
c	$(c, 27, -)$	$(f, 39, n_{17})$
d	$(d, 16, -)$	$(a, 36, n_2)$
e	$(e, 26, -)$	$(d, 38, n_{20})$
f	$(f, 15, -)$	$(a, 19, n_6)$
⋮	⋮	⋮
n_6	$(a, 14, a)$	$(f, 20, n_9)$
⋮	⋮	⋮
n_{19}	$(c, 37, n_{17})$	$(f, 39, n_{17})$
n_{20}	$(d, 29, d)$	$(e, 35, e)$

k times. Finally, the graph $G(N, E)$ with k labels on each node is returned as output (Line 21).

Algorithm 1: Construct-kSPT($\mathcal{D}$, k, $G(N, E)$, q_e)

input : Dataset $\mathcal{D}$, k, Graph $G(N, E)$, Destination q_e
output: Labelled Graph $G(N, E)$

1 Initialize Priority Queue PQ;
2 **for each** (object p in $\mathcal{D}$) **do**
3 Node $n_p \leftarrow$ Create a network node from p;
4 $G(N, E)$.Insert(n_p);
5 Distance $d \leftarrow$ DIST(n_p, q_e);
6 PQEntry $e \leftarrow$ Tuple $(n_p, p, d, -)$;
7 PQ.Insert(e);
8 **for each** (n in $G(N, E)$) **do**
9 n.LabelList $\leftarrow$ Create an empty list of labels;
10 n.Type $\leftarrow$ Labelable;
11 **while** PQ is **not** empty **do**
12 PQEntry $(n, p, d, n_h) \leftarrow PQ$.DequeueHead();
13 **if** n.Type is Labelable **and** no existing label with p **then**
14 n.LabelList.Add((p, d, n_h));
15 **if** n.LabelList.Length is k **then**
16 n.Type $\leftarrow$ Permanent;
17 **for each** Node n_a in n.AdjList that can immediately reach n **and** n_a is Labelable **do**
18 Distance $\omega \leftarrow$ EDGE(n_a, n).Weight;
19 PQEntry $e_a \leftarrow$ Tuple $(n_a, p, d + \omega, n)$;
20 PQ.Insert(e_a);
21 **return** $G(N, E)$;

Let us now consider the first few steps of the algorithm, in the context of the example in Fig. 6. After the initialization steps, the priority queue PQ has the following initial entries:

$$\langle (a, a, 13, -), (f, f, 15, -), (d, d, 16, -), (e, e, 26, -), (c, c, 27, -), (b, b, 32, -) \rangle.$$

The first entry retrieved from PQ is $(a, a, 13, -)$. As a result, Node a is labeled with a itself as the associated detour object and the labeling distance of 13. Next, from the two nodes adjacent to a, n_6, and n_7, two entries $(n_6, a, 14, a)$ and $(n_7, a, 18, a)$ are created, respectively. These entries are inserted into PQ resulting in the following objects in PQ:

$$\langle (n_6, a, 14, a), (f, f, 15, -), (d, d, 16, -), (n_7, a, 18, a), \ldots, (b, b, 32, -) \rangle.$$

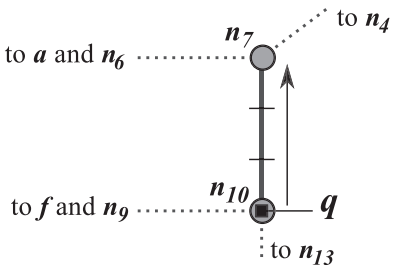


Fig. 7. Starting location of the query point q.

The second retrieved entry $(n_6, a, 14, a)$ has n_6 as the corresponding node, hence we apply the label $(a, 14, a)$ to n_6. The same process continues until PQ is exhausted. The algorithm returns $G(N, E)$ with k labels associated to each node as displayed in Table 3.

The drawback of Algorithm 1 is that it requires access to all data objects and nodes in the data space. This global access requirement can be disadvantageous especially in a large network. In the next section, we show how this drawback can be mitigated. To better illustrate this drawback in comparison to nonglobal access methods, we use Algorithm 1 as a competitor in the experimental studies.

5.2 Incremental kSPT Construction

We now present our CDQ solution which incrementally retrieves data objects and computes node labels as the monitoring process progresses. The computation cost of a kSPT can be greatly reduced by exploiting the fact that the offset assigned to each object p is the distance from p to the destination q_e. Hence, objects that are far away from q_e are likely to be involved in the computation later than objects nearer to q_e. For example, in Fig. 6, when applying the first label $(a, 14, a)$ to n_6, Objects b, c, and e cannot affect the results since their distances to q_e are greater than the labeling distance of 14. Based on this property, we devise a mechanism which is incremental in two aspects:

- *Object retrieval*: through monitoring of the labeling distance and incremental retrieval of data objects;
- *Node labeling*: through an incremental labeling process, which halts when a desired label list is obtained and resumes when more label lists are required.

As a result, we eliminate the global access requirement in terms of both data objects and network nodes. We revisit the running example in Figs. 5 and 6 to elaborate the concept of incremental kSPT construction. As shown in Fig. 7, the query point q is initially at n_{10} (and is traveling toward n_7). The kMDO results of this initial location of q can be obtained from the label list of n_{10}.

We now describe how the label list of n_{10} can be obtained. As the labeling process progresses, detour objects are incrementally retrieved according to their distances to q_e. The scope of this object retrieval (the distance from the farthest retrieved object to q_e) is denoted as a search radius r. The value of r indicates whether a node is safe to label or more detour objects are needed. Fig. 8 presents a stepped explanation to how the k labels of n_{10} can be computed though incremental object retrieval and incremental node labeling.

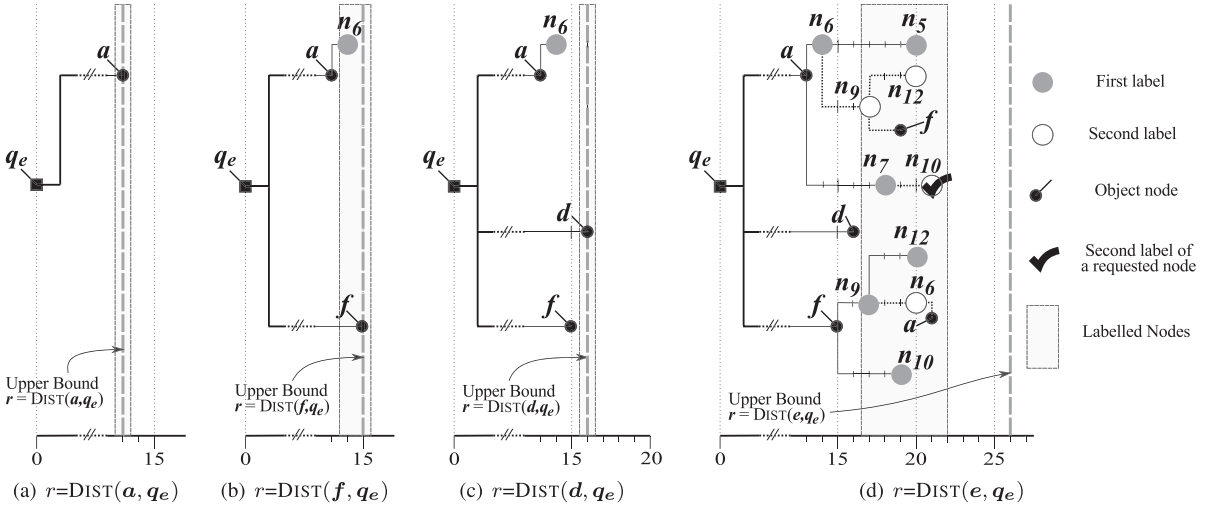


Fig. 8. Incremental kSPT construction with k of 2.

- In Fig. 8a, the first retrieved object is a which is the object nearest to q_e. The search radius r is set to DIST(a, q_e). The only one node that can be labeled with this r value is Node a itself. After the labeling, we move on to consider the next object.
- In Fig. 8b, the next object nearest to q_e is f. The search radius r is updated to DIST(f, q_e), which allows labeling of n_6 and Node f itself, respectively.
- In Fig. 8c, d is retrieved and the value of r is updated DIST(d, q_e). Node d is the only labeled node under this r value.
- In Fig. 8d, e is retrieved. The value of r is updated to DIST(e, q_e), which allows the nodes in the gray region to be labeled. The labeling process halts upon completion of the second label of n_{10}. The figure also shows that n_{10} appears first in the branch of f and again in the branch of a. We can therefore infer that the kMDO list of n_{10} is $\langle f, a\rangle$.

Next, we show how a subsequent label list can be obtained as the query point moves away from the initial location in Fig. 7. Assume that now the query point is at a location on EDGE(n_{10},n_7). In order to produce the kMDO results for this location, we need the label list of n_7 in addition to that of n_{10}. Fig. 9 shows that the second label of n_7 can be obtained by resuming the labeling process halted after the completion of the second label of n_{10}. The figure also shows that we are only required to label n_{13} and n_7 in order to obtain the second label of n_7.

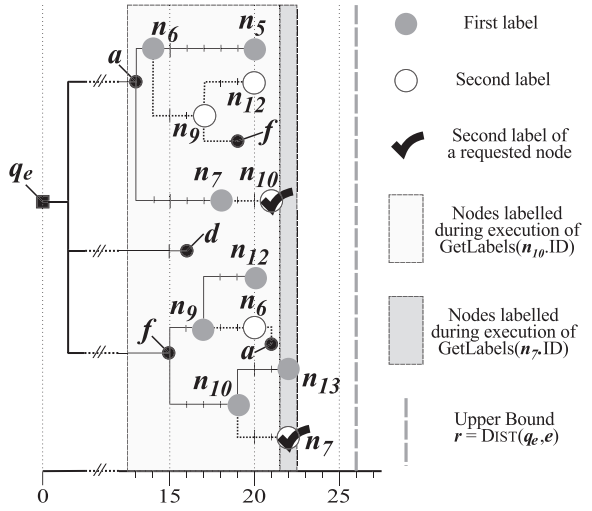


Fig. 9. Incremental kSPT construction with k of 2 (cont.).

Algorithm 2 provides detailed steps of how the label list of a node n_i is obtained. The first step is to check whether the k labels of n_i already exist, in which case the labels are returned right away (Line 1 to 3). For example, after obtaining the label list of n_7 in Fig. 9, the label lists of nodes n_6, n_9, n_{10}, and n_{12} can be obtained without further graph traversal. If the requested label list is otherwise incomplete, we proceed to the main while loop (Lines 4 to 26). The main while loop in Algorithm 2 is similar to that in Algorithm 1. The following modifications are applied to make Algorithm 2 incremental.

- The first modification is the search radius check (Lines 6 to 12), which ensures that the value of r is not smaller than the labeling distance d. Specifically, until r is greater than or equal to d, the following steps are repeated:
  - retrieving the next NN with respect to q_e (Line 7);[3]
  - performing graph modification and priority queue insertion (Lines 8 to 11) similar to Algorithm 1;
  - setting the search radius r to the distance from that object to q_e (Line 12).
- The second modification is deferral of node initialization (Lines 15 to 17).
- The final modification is a halt to the node labeling process after the requested node has k labels (Lines 25 and 26).

Algorithm 3 provides detailed steps of how kMDO monitoring can be conducted using Algorithm 2. Algorithm 3 has the following parameters: a data set $\mathcal{D}$, a k value, a road-network graph $G(N, E)$, and a destination q_e. The initialization steps (Lines 1 to 7) include:

3. We omit presentation of a marginal case where the data set is exhausted, in which case we set r to infinity and break the loop.

- initializing the priority queue PQ;
- retrieving the nearest object to $\boldsymbol{q}_e$ as the initial object $\boldsymbol{p}$, and inserting it into $G(N,E)$ and PQ;
- setting the search radius r to DIST($\boldsymbol{p}, \boldsymbol{q}_e$).

The kMDO monitoring process is conducted in the while loop (Lines 9 to 12). In the context of our running example, Algorithm 3 uses Algorithm 2 to produce the label lists of $\boldsymbol{n}_{10}$ and $\boldsymbol{n}_7$: $\langle(\boldsymbol{f},19,\boldsymbol{f}),\ (\boldsymbol{a},21,\boldsymbol{n}_7)\rangle$, and $\langle(\boldsymbol{a},18,\boldsymbol{a}),\ (\boldsymbol{f},22,\boldsymbol{n}_{10})\rangle$, respectively. These label lists are then used to produce the kMDO results for the current location of the query point (Line 11). These results are then reported to the user (Line 12). In the next section, we show how the kMDO of any location in $G(N,E)$ can be derived from such label lists.

Algorithm 2: GetLabelList(NodeID)

environment: Dataset $\mathcal{D}$, k, Graph $G(N,E)$, PQ, r
input : NodeID
output : LabelList $\langle l_1, ..., l_k\rangle$

1 $\boldsymbol{n} \leftarrow$ GetNode(NodeID, $G(N,E)$);
2 **if** $\boldsymbol{n}$ has been initialized **and** $\boldsymbol{n}$.Type is Permanent **then**
3 **return** $\boldsymbol{n}$.LabelList;
4 **while** PQ is **not** empty **do**
5 PQEntry $(\boldsymbol{n}, \boldsymbol{p}, d, \boldsymbol{n_h}) \leftarrow PQ$.Head();
6 **while** $r < d$ **do**
7 Object $\boldsymbol{p} \leftarrow$ GetNextNN($\boldsymbol{q_e}$,$\mathcal{D}$);
8 Node $\boldsymbol{n_p} \leftarrow$ Create a network node from $\boldsymbol{p}$;
9 $G(N,E)$.Insert($\boldsymbol{n_p}$);
10 PQEntry $e \leftarrow$ Tuple $(\boldsymbol{n_p}, \boldsymbol{p}, \text{DIST}(\boldsymbol{p}, \boldsymbol{q_e}), -)$;
11 PQ.Insert(e);
12 $r \leftarrow \text{DIST}(\boldsymbol{p}, \boldsymbol{q_e})$
13 PQEntry $(\boldsymbol{n}, \boldsymbol{p}, d, \boldsymbol{n_h}) \leftarrow PQ$.DequeueHead();
14 **if** $\boldsymbol{n}$.Type is Labelable **and** no existing label with $\boldsymbol{p}$ **then**
15 **if** $\boldsymbol{n}$ is **not** initialized **then**
16 $\boldsymbol{n}$.LabelList $\leftarrow$ Create an empty list of labels;
17 $\boldsymbol{n}$.Type $\leftarrow$ Labelable;
18 $\boldsymbol{n}$.LabelList.Add($(\boldsymbol{p}, d, \boldsymbol{n_h})$);
19 **for each** Node $\boldsymbol{n_a}$ in $\boldsymbol{n}$.AdjList that can immediately reach $\boldsymbol{n}$ **and** $\boldsymbol{n_a}$ is Labelable **do**
20 Distance $\omega \leftarrow \text{EDGE}(\boldsymbol{n_a}, \boldsymbol{n})$.Weight;
21 PQEntry $e_a \leftarrow$ Tuple $(\boldsymbol{n_a}, \boldsymbol{p}, d+\omega, \boldsymbol{n_h})$;
22 PQ.Insert(e_a);
23 **if** $\boldsymbol{n}$.LabelList.Length is k **then**
24 $\boldsymbol{n}$.Type $\leftarrow$ Permanent;
25 **if** $\boldsymbol{n}$.ID = NodeID **then**
26 **return** $\boldsymbol{n}.\langle L_1, ..., L_k\rangle$;

Algorithm 3: IkSPT-CDQ ($\mathcal{D}$, k, $G(N,E)$, $\boldsymbol{q_e}$)

environment: Trajectory T containing locations $(\boldsymbol{n_i}, \boldsymbol{n_j}, \alpha)$
input : $\mathcal{D}$, k, $G(N,E)$, $\boldsymbol{q_e}$
output : Reporting kMDO results at each location

1 Initialize Priority Queue PQ;
2 $\boldsymbol{p} \leftarrow$ GetNN($\boldsymbol{q_e}$,$\mathcal{D}$);
3 $r \leftarrow \text{DIST}(\boldsymbol{p}, \boldsymbol{q_e})$;
4 Node $\boldsymbol{n_p} \leftarrow$ Create a network node from $\boldsymbol{p}$;
5 $G(N,E)$.Insert($\boldsymbol{n_p}$);
6 PQEntry $e \leftarrow$ Tuple $(\boldsymbol{n_p}, \boldsymbol{p}, \text{DIST}(\boldsymbol{n_p}, \boldsymbol{q_e}), -)$;
7 PQ.Insert(e);
8 **while** More Location $(\boldsymbol{n_i}, \boldsymbol{n_j}, \alpha)$ *in* T **do**
9 LabelList $L_i \leftarrow$ GetLabelList($\boldsymbol{n_i}$.ID);
10 LabelList $L_j \leftarrow$ GetLabelList($\boldsymbol{n_j}$.ID);
11 Results $\mathcal{A} \leftarrow$ GetResults(EDGE($\boldsymbol{n_i}$,$\boldsymbol{n_j}$), L_i, L_j, α);
12 Report($\mathcal{A}$);

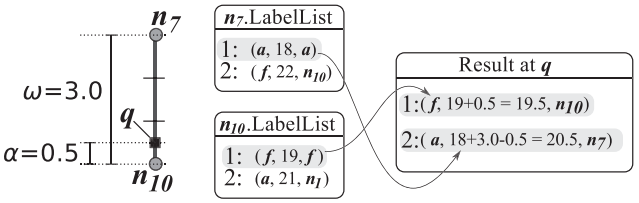


Fig. 10. Derivation of kMDO results from two label lists.

5.3 Derivation of kMDO from Node Labels

In the previous section, we have shown that the kMDO of any node in the network can be incrementally obtained using Algorithm 2. We now present a method to calculate the kMDO list of any location on an edge. For a query point $\boldsymbol{q}$ on a bidirectional edge EDGE($\boldsymbol{n_i}$,$\boldsymbol{n_j}$), the kMDO list of $\boldsymbol{q}$ may comprise objects from the lists of both end. (For a directional edge EDGE($\boldsymbol{n_i}, \boldsymbol{n_j}$), the kMDO list of $\boldsymbol{q}$ is the same as that of $\boldsymbol{n_i}$ if and only if $\boldsymbol{q}$ is exactly at $\boldsymbol{n_i}$. Otherwise, the kMDO list is the same as that of $\boldsymbol{n_j}$, since $\boldsymbol{q}$ cannot leave EDGE($\boldsymbol{n_i}, \boldsymbol{n_j}$) without passing $\boldsymbol{n_j}$.[4])

We use Fig. 10 to illustrate how the kMDO list can be obtained in such a case. Let ω denote the edge's weight and α denote the distance from $\boldsymbol{n_i}$ to the query point $\boldsymbol{q}$ along the edge. In this example, the query point $\boldsymbol{q}$ is on EDGE($\boldsymbol{n_{10}}, \boldsymbol{n_7}$) with α of 0.5 units and ω of 3.0 units. The (order sensitive) kMDO lists of $\boldsymbol{n_{10}}$ and $\boldsymbol{n_7}$ are $\langle \boldsymbol{f}, \boldsymbol{a}\rangle$ and $\langle \boldsymbol{a}, \boldsymbol{f}\rangle$, respectively. For each unique object $\boldsymbol{p}$ in those lists, TRIPDIST($\boldsymbol{q}, \boldsymbol{p}, \boldsymbol{q_e}$) can be obtained by comparing the distance via $\boldsymbol{n_{10}}$ and the distance via $\boldsymbol{n_7}$. Specifically, we select the node that provides the minimum distance of (TRIPDIST($\boldsymbol{n_{10}}, \boldsymbol{p}, \boldsymbol{q_e}$) + α) and (TRIPDIST($\boldsymbol{n_7}, \boldsymbol{p}, \boldsymbol{q_e}$) + $\omega - \alpha$). Applying this principle to $\boldsymbol{a}$ and $\boldsymbol{f}$, we can see that:

- $\boldsymbol{n_{10}}$ provides the minimum distance of 19.5 units for $\boldsymbol{f}$;
- $\boldsymbol{n_7}$ provides the minimum distance of 20.5 units for $\boldsymbol{a}$.

Algorithm 4 provides a formal description of the result derivation process. The input parameters are 1) the edge EDGE($\boldsymbol{n_i}, \boldsymbol{n_j}$) on which the query point resides, 2) the two labels L_i and L_j of the end nodes, and 3) the distance offset α indicating the location of the query point relative to $\boldsymbol{n_i}$. Lines 7 to 21 handle the case where EDGE($\boldsymbol{n_i}, \boldsymbol{n_j}$) is bidirectional. (In Line 1, we check whether the edge is directional. If so, we return L_i if the query point is right on $\boldsymbol{n_i}$. Otherwise, L_j is returned, since $\boldsymbol{n_i}$ is reachable only via $\boldsymbol{n_j}$ in this case.) For each unique object $\boldsymbol{p}$ from L_i and L_j, we compare the distance d_i via $\boldsymbol{n_i}$ and the distance d_j via $\boldsymbol{n_j}$. If there is no entry with $\boldsymbol{p}$ in L_i, the distance d_i is infinity (i.e., ignored). This is because in such a case, the actual d_i cannot affect the kMDO list of $\boldsymbol{q}$. The same logic also applies to d_j. The minimum of d_i and d_j forms TRIPDIST($\boldsymbol{q}, \boldsymbol{p}, \boldsymbol{q_e}$). Next, a label is created with the MDO of $\boldsymbol{p}$, the labeling distance of TRIPDIST($\boldsymbol{q}, \boldsymbol{p}, \boldsymbol{q_e}$). The label is inserted into the list $\mathcal{A}$. After the TRIPDISTs of all unique objects are calculated, the top k resultant labels in $\mathcal{A}$ are returned.

4. Assume no objects on EDGE($\boldsymbol{n_i}, \boldsymbol{n_j}$).

Algorithm 4: GetResults(EDGE(n_i,n_j), L_i, L_j, α)

```
input  : EDGE(n_i,n_j), LabelList L_i, LabelList L_j, Distance α
output : LabelList ⟨l_1, ..., l_k⟩
1  if EDGE(n_i,n_j) is directional then
2      if α = 0 then
3          return L_i
4      else
5          return L_j
6  else
7      Initialize a list A of resultant labels;
8      for each unique object p from L_i and L_j do
9          Initialize d_i and d_j to infinity;
10         if there exists a label with p in L_i then
11             (p, d, n_h) ← Label with p in L_i;
12             d_i ← d + α;
13         if there exists a label with p in L_j then
14             (p, d, n_h) ← Label with p in L_j;
15             d_j ← d + EDGE(n_i, n_j).Weight − α;
16         if d_i ≤ d_j then
17             A.Insert(p, d_i, n_i);
18         else
19             A.Insert(p, d_j, n_j);
20     Sort(A);
21     return Head(k, A)
```

The correctness of Algorithm 4 is guaranteed by Lemma 1. Specifically, the lemma states that the kMDO of a location between two nodes of a bidirectional edge can be obtained from the information given by the labels of those two nodes.

Lemma 1. *Given an edge EDGE(n_i, n_j) and a common destination q_e, let $\mathcal{A}_i$ and $\mathcal{A}_j$ be the kMDO lists of n_i and n_j, respectively. If an object p is one of the kMDOs of any point v on EDGE(n_i, n_j), then p is a member of $\mathcal{A}_i$ and/or $\mathcal{A}_j$.*

Proof. We prove this lemma by showing that for any object p that is neither a member of $\mathcal{A}_i$ nor $\mathcal{A}_j$, there are at least k objects r in $\mathcal{A}_i$ or $\mathcal{A}_j$ such that

$$\text{TRIPDIST}(v, p, q_e) \geq \text{TRIPDIST}(v, r, q_e).$$

For any point v on EDGE(n_i, n_j), TRIPDIST(v, p, q_e) is minimum between (DIST(v, n_i) + TRIPDIST(n_i, p, q_e)) and (DIST(v, n_j) + TRIPDIST(n_j, p, q_e)). We therefore split derivation of TRIPDIST(v, p, q_e) into two cases. First, if n_i provides a TRIPDIST less than or equal to that of n_j, then

$$\text{TRIPDIST}(v, p, q_e) = \text{DIST}(v, n_i) + \text{TRIPDIST}(n_i, p, q_e).$$

Let p_i be any object in $\mathcal{A}_i$. If p is not in $\mathcal{A}_i$, then

$$\text{TRIPDIST}(n_i, p, q_e) \geq \text{TRIPDIST}(n_i, p_i, q_e).$$

This inequality implies that TRIPDIST(v, p, q_e) must be greater than or equal to that of any of the k objects in $\mathcal{A}_i$.

Second, when n_j provides a smaller TRIPDIST, the same principle can be applied to show that TRIPDIST(v, p, q_e) is not less than that of any of the k objects in $\mathcal{A}_j$.

These two cases imply that for any given point v on EDGE(n_i,n_j), there must be at least k detour objects from either $\mathcal{A}_i$ or $\mathcal{A}_j$ that provide TRIPDISTs smaller than or equal to TRIPDIST(v, p, q_e), and hence p can be safely ignored as a query result at v. □

Note that by considering data objects as network nodes, the kMDO results of any location on EDGE(n_i, n_j) can be derived from those of n_i and n_j. However, in a setting where such graph modification is inapplicable, the same principle can still be applied. In this case, to produce the k MDOs of any location q on EDGE(n_i, n_j), we need to also consider detour objects along EDGE(n_i, n_j) in addition to the kMDO results at the end nodes.

We can observe that the kMDO of any location on an edge EDGE(n_i, n_j) can be efficiently computed by comparing the distances of entries in the label lists of n_i and n_j. Alternatively, one may apply a kNN monitoring algorithm (e.g., the split point calculation algorithm [5] or the incremental rank update algorithm [12]) to identify points along EDGE(n_i, n_j) where the kMDO changes. Using these split points, an edge can be decomposed into intervals where each corresponds to a particular kMDO. Consequently, we can replace repetitive distance comparison and sorting operations by boundary check and incremental update operations, respectively. Implementation of these techniques is independent of the proposed IkSPT construction algorithm.

6 Competitive Method: Incremental Network Expansion

In this section, we formulate a competitive method based on the existing concept of AkNN querying [19], [26]. In this method, we apply the *multiple query method* [19] to retrieve k MDOs with respect to a starting location q_s and a destination q_e. We call this method INE-CDQ, since we use the *INE* principle to retrieve

1. NNs from q_s (objects p that minimize DIST(q_s, p));
2. NNs to q_e (objects p that minimize DIST(p, q_e)).

The order in which these objects are retrieved depends on the search coverage C_s with respect to q_s and the search coverage C_e with respect to q_e. That is, the next NN with respect to q_s is retrieved if C_s is smaller than C_e and vice versa. Based on these coverages, each retrieved object p is categorized into two types.

- *Candidate*: p is discovered via both q_s and q_e.
- *Precandidate*: p is discovered via either q_s or q_e only.

The TRIPDIST of each candidate p is the sum of DIST(q_s, p) and DIST(p, q_e), while a lower bound is used to represent the smallest possible TRIPDIST produced by all precandidates.

Let p_s and p_e be the nearest precandidates discovered by q_s and q_e, respectively. A TRIPDIST lower bound L of all precandidates is calculated by substituting the search coverages C_s and C_e for the unknown distances, i.e.,

$$L = \text{MIN}\{\text{DIST}(q_s, p) + C_e, C_s + \text{DIST}(p, q_e)\}.$$

Assume that all candidates p are maintained in a priority queue PQ_c, where entries are organized according to TRIPDIST(q_s, p, q_e). The head entry of PQ_c is the next MDO with respect to q_s and q_e if its TRIPDIST is smaller than the lower bound L. Otherwise, the next NN has to be retrieved in order to determine the next MDO. The process continues until k MDOs are discovered.

Since this method constructs kMDO results based on a specific query location, the results have to be reevaluated as

the query point q_s encounters an unvisited node. However, we can exploit the fact that q_e is a fixed destination and reuse the network expansion result for subsequent kMDO evaluations. That is, all objects discovered via q_e are cached and the SPT with respect to q_e is incrementally expanded per demand. As a result, each node is visited by the network expansion with respect to q_e at most once.

Note that one can also modify TPQ and OSR to find k MDOs with respect to a query location q_s and a destination q_e by limiting the sequence length to 2, where 1) the first element is a detour object p, and 2) the second element is a fixed destination q_e. We can then find objects p that minimize the total sequence length of $(\text{DIST}(q_s, p) + \text{DIST}(p, q_e))$. As can be seen, this modification results with an approach comparable to AkNN.

7 Discussion on Proposed and Competitive Methods

In this section, we discuss the advantages and drawbacks of our proposed method IkSPT − CDQ (Algorithm 3), in comparison to the following two competitive techniques.

- *INE-CDQ*: incremental network expansion around q_s and q_e (as described in Section 6).
- kSPT-CDQ: construction of an entire order-k shortest path tree (as shown in Algorithm 1).

INE-CDQ applies the incremental network expansion principle to retrieve kMDO with respect to the current location of the query point q_s. Since such an expansion is *query-centered*, the distances from q_s to its surrounding nodes have to be reevaluated from scratch when q_s moves. As shown in Algorithm 1 (kSPT-CDQ), we can avoid repetitive distance evaluation by applying network expansion in a *data-centered* manner. Specifically, network distances to a set $\mathcal{D}$ of data objects are calculated by using each detour object p in $\mathcal{D}$ as a center of expansion (where p is weighted by $\text{DIST}(p, q_e)$). Algorithm 1 terminates only when each node n is associated with k detour objects p with the smallest $\text{TRIPDIST}(n, p, q_e)$. As a result, kSPT-CDQ is disadvantageous to INE-CDQ when the query locations are close to the destination in relative to the size of the network.

In our proposed method, IkSPT-CDQ, we exploit a special property of the CDQ problem where data objects are additively weighted based on their distances to the destination q_e. As a result, the order in which nodes are visited by network expansion is skewed toward q_e. Based on this knowledge, IkSPT-CDQ is formulated by allowing the labeling process to halt when the requested node is complete and to resume when more kMDO results are required. Therefore, the query processing cost depends on the distance from the query location q_s to the destination q_e. In this respect, IkSPT-CDQ has a similar behavior to INE-CDQ. This is because, in order to retrieve the first MDO with respect to given locations of q_s and q_e, INE-CDQ has to consider at least objects in the search space of

$$\{v : \text{DIST}(q_s, v) < d/2\} \cup \{v : \text{DIST}(v, q_e) < d/2\},$$

where d denotes $\text{DIST}(q_s, q_e)$. However, IkSPT-CDQ does not suffer from repetitive result evaluation, since kMDO results of each node are obtained via data-centric expansion. To provide a more comprehensive cost comparison between the three methods, experimental results are reported in the next section.

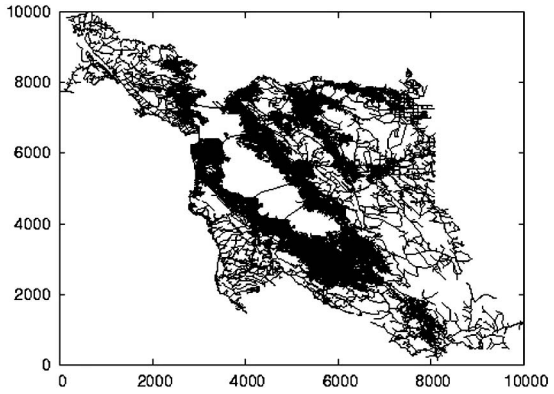


Fig. 11. Main roads around the San Francisco bay area.

8 Experiments

This section presents experimental studies on the two competitors, *INE-CDQ* and *kSPT-CDQ*, and our proposed method, IkSPT-CDQ. Our experiments are conducted on a 2.66 GHz Intel Core 2 Duo machine with 4.0 GB of main memory. All algorithms are implemented in Java. As displayed in Fig. 11, we use a large road network representing main roads in and around San Francisco. The network consists of 174,956 nodes and has a network diameter of 16,824 units.

8.1 Experimental Setup

Our experimental setup is based on an application scenario where a user at the starting location q_s wishes to travel to a location q_e. The user also would like to monitor a list of k MDOs within a set $\mathcal{D}$ of detour objects. After traveling for l units, the user selects a detour object and stops monitoring the k MDOs. Therefore, no further monitoring is needed and the query can terminate.

We use two performance measures: 1) the total execution time; 2) the graph traversal cost. The total execution time is measured as the amount of time a technique uses to process one trajectory. The graph traversal cost is measured as the number of times network nodes are accessed through kSPT construction and network distance calculations.

Our experimental studies include the following parameters:

- the monitoring distance l along the query trajectory,
- the distance d from the original query location of q_s and the destination q_e,
- the number k of requested MDOs, and
- the relative density ρ of nodes with respect to objects.

Table 4 provides the details of these parameters. The default values and ranges of these experimental parameters are derived from the aforementioned application scenario. We assume that a user (while moving) is capable of paying attention to only a small number of MDOs so the range of k is set to [2, 10]. The range of ρ is set to [100, 500] to emulate the number of nodes per point of interest (e.g., a restaurant or a

TABLE 4
Experimental Parameters

Parameter	Default	Min	Max	Increment
l	400	0	800	200
k	6	2	10	2
d	2,400	800	4,000	800
ρ	300	100	500	100

hotel) that matches the user's preference. That is, in a network with 174,956 nodes, the number of objects is ranged from 350 to 1,750 objects. We use a high number of nodes per object, since data objects that match user's preference are more realistically represented this way. For example, there maybe hundreds of restaurants of a particular cuisine and price range, although in total there maybe thousands of restaurant in one city.[5]

The range of d is set to [800, 4,000] to represent typical traveling distances in one city, e.g., 5 to 25 percent of the diameter of the San Francisco network. The default kMDO monitoring distance l and the default value of the distance d from the origin to the destination are selected to emulate a user making their detour decision early in their trip. That is, the l value of 400 units and the distance d from $\boldsymbol{q_s}$ to $\boldsymbol{q_e}$ of 2,400 units correspond to the user making their detour decision after traveling 1/6 of the distance to the destination $\boldsymbol{q_e}$. Assume that the user intends to drive to a destination 30 miles away from the origin and is traveling at a constant speed of 45 mph. The detour decision is made within the first 6-7 minutes of the trip.

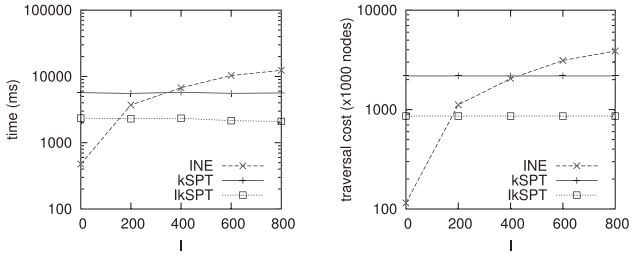
We use two types of trajectories: *directional* and *random*. For each type, we generated 10 instances as a query set. Experimental results are reported as the average of results from these trajectories. We used the same set of starting locations $\boldsymbol{q_s}$ for both types. A directional trajectory was generated as the shortest path $\text{PATH}(\boldsymbol{q_s},\boldsymbol{q_e})$ with a starting point $\boldsymbol{q_s}$. The trajectory is terminated at the location $\boldsymbol{v}$ on $\text{PATH}(\boldsymbol{q_s},\boldsymbol{q_e})$ such that $\text{DIST}(\boldsymbol{q_s},\boldsymbol{v})$ is equal to l. A random trajectory was generated as a sequence of subtrajectories (where the end of one subtrajectory is the starting location of the next one). Each subtrajectory is a shortest path to a random destination (independent of $\boldsymbol{q_e}$) and has a length of 100 units. The trajectory length l is measured as the total length of all subtrajectories. For example, a trajectory with a length of 400 units contains four subtrajectories.

These two trajectory types represent two common cases. First, a directional trajectory represents a case where the directionality is absolute, e.g., a truck driver follows the shortest path to a destination while keeping track of a pizza place to get some food. Second, a random trajectory represents a case where the directionality is weak, e.g., a tourist visits sightseeing destinations while keeping track of supermarkets to drop by before going back to her hotel.

8.2 Experimental Results

8.2.1 Effect of l on INE, kSPT, and IkSPT

In the first set of experiments, we show the effect of the monitoring distance l on the execution time and traversal cost of the three methods. The value of l is ranged from 0 to 800 units. The l value of 0 corresponds to the query being used as a snapshot query, i.e., finding an instantaneous result without continuous monitoring. According to Figs. 12 and 13, INE is the best method when used for a snapshot query. However, when the monitoring distance l increases, execution time and traversal cost of INE consistently increase for directional and random trajectories. This is because, INE has to reevaluate kMDO results for each unvisited node encountered by the query point.



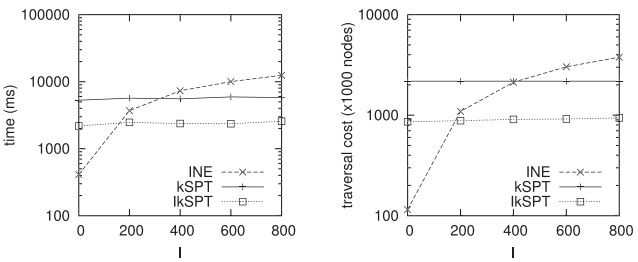
(a) Execution time (directional) (b) Traversal cost (directional)

Fig. 12. Effect of l (directional trajectories).

For kSPT, on the other hand, both execution time and traversal cost remain unchanged, because kSPT computes the kMDO results for all nodes in the network. For IkSPT, changes in l do not produce a noticeable effect on both cost measures. This is because while computing the kMDO results for the initial query location of $\boldsymbol{q_s}$, IkSPT also computes results or partial results for its surrounding node. Hence, the incremental cost of computing subsequent node is negligible. The experimental results show that INE is the best method for a snapshot query, while IkSPT is the best method if continuous kMDO monitoring is required.

8.2.2 Effect of d on INE, $kSPT$, and $IkSPT$

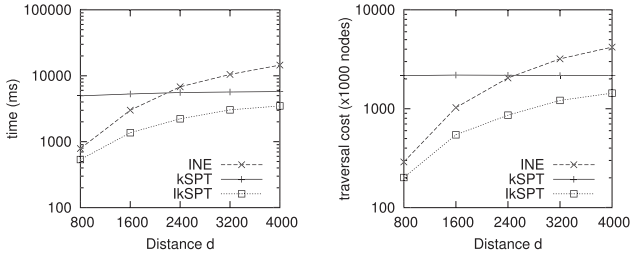
In the second set of experiments, we show how the three methods scale as the distance d from the initial location $\boldsymbol{q_s}$ to destination $\boldsymbol{q_e}$ increases. The distance d is varied from 800 to 4,000 units. According to Figs. 14 and 15, since kSPT computes kMDO results for all nodes in the network, the distance d has no effect on either the execution time or the traversal cost for both directional and random trajectories. For INE and IkSPT, on the other hand, the distance d has positive correlation with their execution time and traversal cost. These results conform with our discussion in Section 7. We can also see that IkSPT has a much smaller cost and scales better than INE as d increases.



(a) Execution time (random) (b) Traversal cost (random)

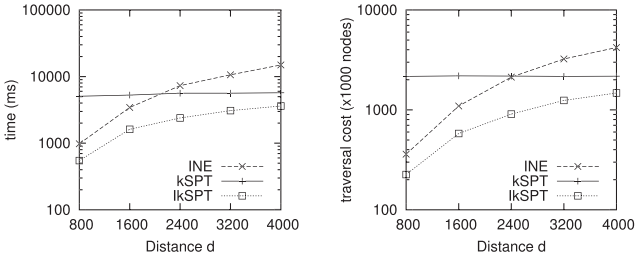
Fig. 13. Effect of l (random trajectories).

5. Objects that do not match the user's preference can be pruned during data retrieval (Line 7 of Algorithm 2 and Line 2 of Algorithm 3). Hence, they are not involved in kMDO labeling of nodes.



(a) Execution time (directional) (b) Traversal cost (directional)

Fig. 14. Effect of d (directional trajectories).



(a) Execution time (random) (b) Traversal cost (random)

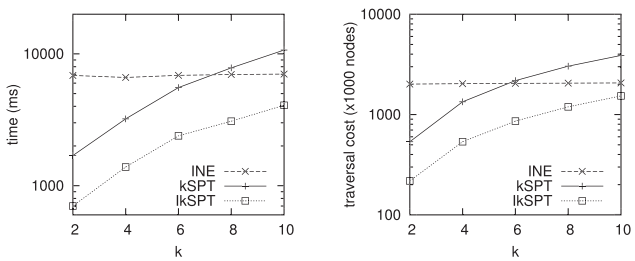
Fig. 15. Effect of d (random trajectories).

8.2.3 Effect of k on INE, $kSPT$, and $IkSPT$

The third set of experiments presents a study on the effect of the number k of requested MDOs on the three methods. The value of k is varied from 2 to 10. According to Fig. 16 (directional) and Fig. 17 (random), the parameter k has no noticeable effect on the execution time or the traversal cost of INE. As explained in Section 7, each MDO has to be covered by the network expansions with respect to $\boldsymbol{q_s}$ and $\boldsymbol{q_e}$. This requirement creates a high setup cost in order to determine the first MDO. This setup cost dominates the incremental cost of determining the subsequent $(k-1)$ MDOs. As k increases, kSPT and IkSPT have similar behavior. That is, both execution time and traversal cost increase as k increase. This is because, k determines the number of labels for each node. Since kSPT has to construct the kMDO results for all nodes in the network and IkSPT halts when requested nodes are complete, The execution time and traversal cost of IkSPT are much smaller than those of kSPT.

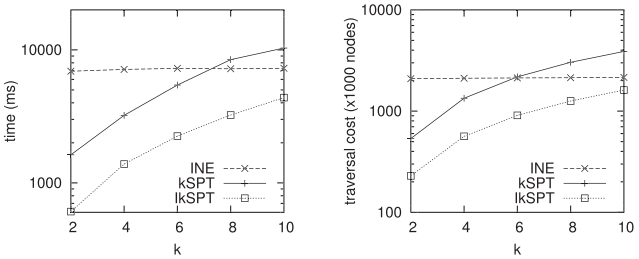
8.2.4 Effect of ρ on INE, $kSPT$, and $IkSPT$

In the last set of experiments, we vary the parameter ρ from 100 to 500 nodes per object. According to Figs. 18 and 19, for kSPT and IkSPT, we see slight increases in the execution time and traversal cost as ρ increases. We have found that as



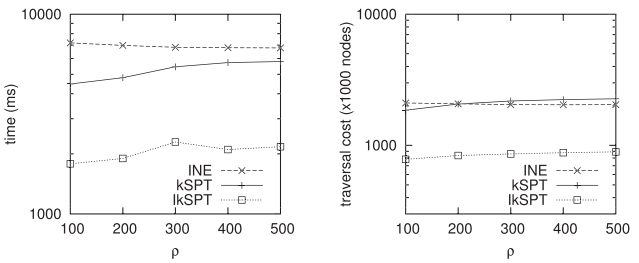
(a) Execution time (directional) (b) Traversal cost (directional)

Fig. 16. Effect of k (directional trajectories).



(a) Execution time (random) (b) Traversal cost (random)

Fig. 17. Effect of k (random trajectories).



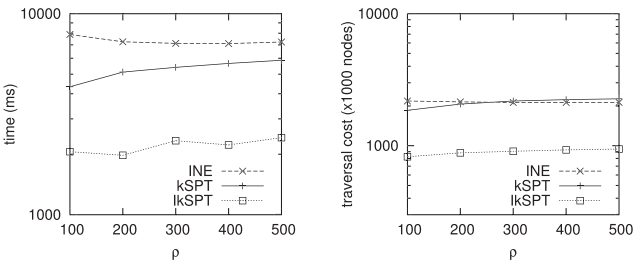
(a) Execution time (directional) (b) Traversal cost (directional)

Fig. 18. Effect of ρ (directional trajectories).

the number of detour objects reduces (as a result of an increase in the number ρ of nodes per object), each kSPT branch (corresponding to a detour object) becomes larger and results in a greater number of priority queue entries. Hence, as ρ increases, we can see slight increases in the execution time and traversal cost of kSPT and IkSPT for both directional and random trajectories (Figs. 18 and 19). For INE, since the minimum search space is determined by the distance from $\boldsymbol{q_s}$ to $\boldsymbol{q_e}$, there is no correlation between ρ and the traversal cost. The execution time of INE, on the other hand, decreases and ρ increases. This is because as ρ increases the number of objects reduces, which produces less candidates and precandidates to process in order to determine the k MDOs. We can also see that IkSPT continues to outperform the other two methods in both cost measures.

8.3 Summary

We have derived ranges of experimental parameters and their default values based on a realistic CDQ setting. Since IkSPT is a localization of kSPT, IkSPT can only perform better than or similar to kSPT for both cost measures. IkSPT scales better than INE as the distance d from $\boldsymbol{q_s}$ to $\boldsymbol{q_e}$ and the monitoring distance l increase. Although IkSPT scales



(a) Execution time (random) (b) Traversal cost (random)

Fig. 19. Effect of ρ (random trajectories).

worse than INE as k increases, IkSPT still performs better than INE for all values of k in the range of [2, 10].

9 Conclusions

We formulated the *CDQ* and proposed an efficient method to process the query. We have compared our proposed method, IkSPT-CDQ, to two competitive techniques, INE-CDQ and kSPT-CDQ. INE-CDQ constructs kMDO results on a node-by-node basis and uses the Dijkstra's algorithm [9] to explore detour objects around the current query location. When the query point encounters a new node, the kMDO results have to be reevaluated. The other competitive method, kSPT-CDQ, constructs kMDO results in a data-centered manner and calculates kMDO results for all nodes in the network.

Our proposed method, IkSPT-CDQ, incrementally retrieves data objects according to their distances to the destination and computes kMDO results for a subset of nodes in the network. As a result, IkSPT-CDQ does not incur access to all objects and network nodes. Since our method enables construction of kSPTs during runtime, one can apply preconditions to the object retrieval process in order to exclude irrelevant objects from IkSPT construction.

Experimental results show that for the default values of parameters, IkSPT-CDQ is 3.1 times as fast as INE-CDQ and 2.3 times as fast as kSPT-CDQ. In terms of the traversal cost, IkSPT-CDQ has an improvement factor of 2.3 times and 2.4 times in comparison to INE-CDQ and kSPT-CDQ, respectively.

Acknowledgments

This work was supported in part by the Australian Research Council's Discovery funding scheme under Grant DP0880215, and the US National Science Foundation (NSF) under Grants IIS-08-12377 and CCF-08-30618.

References

[1] P.H. Bloch, N.M. Ridgway, and D.L. Sherrell, "Extending the Concept of Shopping: An Investigation of Browsing Activity," *J. Academy of Marketing Science,* vol. 17, no. 1, pp. 13-21, 1989.

[2] M. Brown, N. Pope, and K. Voges, "Buying or Browsing?: An Exploration of Shopping Orientations and Online Purchase Intention," *European J. Marketing,* vol. 37, no. 11, pp. 1666-1684, 2003.

[3] M.A. Cheema, L. Brankovic, X. Lin, W. Zhang, and W. Wang, "Multi-Guarded Safe Zone: An Effective Technique to Monitor Moving Circular Range Queries," *Proc. IEEE 26th Int'l Conf. Data Eng. (ICDE),* pp. 189-200, 2010.

[4] Z. Chen, H.T. Shen, X. Zhou, and J.X. Yu, "Monitoring Path Nearest Neighbor in Road Networks," *Proc. 35th ACM SIGMOD Int'l Conf. Management of Data,* pp. 591-602, 2009.

[5] H.-J. Cho and C.-W. Chung, "An Efficient and Scalable Approach to CNN Queries in a Road Network," *Proc. 31st Int'l Conf. Very Large Data Bases (VLDB),* pp. 865-876, 2005.

[6] A. Corral, Y. Manolopoulos, Y. Theodoridis, and M. Vassilakopoulos, "Closest Pair Queries in Spatial Databases," *Proc. ACM SIGMOD Int'l Conf. Management of Data,* pp. 189-200, 2000.

[7] U. Demiryurek, F.B. Kashani, and C. Shahabi, "Efficient Continuous Nearest Neighbor Query in Spatial Networks Using Euclidean Restriction," *Proc. 11th Int'l Symp. Advances in Spatial and Temporal Databases (SSTD),* pp. 25-43, 2009.

[8] K. Deng, X. Zhou, H.T. Shen, S.W. Sadiq, and X. Li, "Instance Optimal Query Processing in Spatial Networks," *VLDB J.,* vol. 18, no. 3, pp. 675-693, 2009.

[9] E.W. Dijkstra, "A Note on Two Problems in Connection with Graphs," *Numeriche Mathematik,* vol. 1, pp. 269-271, 1959.

[10] M.R. Kolahdouzan and C. Shahabi, "Voronoi-Based k Nearest Neighbor Search for Spatial Network Databases," *Proc. 30th Int'l Conf. Very Large Data Bases (VLDB),* pp. 840-851, 2004.

[11] M.R. Kolahdouzan and C. Shahabi, "Alternative Solutions for Continuous k Nearest Neighbor Queries in Spatial Network Databases," *GeoInformatica,* vol. 9, no. 4, pp. 321-341, 2005.

[12] L. Kulik and E. Tanin, "Incremental Rank Updates for Moving Query Points," *Proc. Int'l Conf. Geographic Information Science (GIScience),* pp. 251-268, 2006.

[13] F. Li, D. Cheng, M. Hadjieleftheriou, G. Kollios, and S.-H. Teng, "On Trip Planning Queries in Spatial Databases," *Proc. Ninth Int'l Symp. Advances in Spatial and Temporal Databases (SSTD '05),* pp. 273-290, 2005.

[14] W.W. Moe, "Buying, Searching, or Browsing: Differentiating between Online Shoppers Using in-Store Navigational Clickstream," *J. Consumer Psychology,* vol. 13, no. 1, pp. 29-39, 2003.

[15] K. Mouratidis, M.L. Yiu, D. Papadias, and N. Mamoulis, "Continuous Nearest Neighbor Monitoring in Road Networks," *Proc. 32nd Int'l Conf. Very Large Data Bases (VLDB),* pp. 43-54, 2006.

[16] S. Nutanong, R. Zhang, E. Tanin, and L. Kulik, "Analysis and Evaluation of V*-kNN: An Efficient Algorithm for Moving kNN Queries," *VLDB J.,* vol. 19, no. 3, pp. 307-332, 2010.

[17] A. Okabe, B. Boots, K. Sugihara, and S.N. Chiu, *Spatial Tessellations: Concepts and Applications of Voronoi Diagrams,* second ed. Wiley, 2000.

[18] A. Okabe, T. Satoh, T. Furuta, A. Suzuki, and K. Okano, "Generalized Network Voronoi Diagrams: Concepts, Computational Methods, and Applications," *Int'l J. Geographical Information Science,* vol. 22, no. 9, pp. 965-994, 2008.

[19] D. Papadias, Y. Tao, K. Mouratidis, and C.K. Hui, "Aggregate Nearest Neighbor Queries in Spatial Databases," *ACM Trans. Database Systems,* vol. 30, no. 2, pp. 529-576, 2005.

[20] D. Papadias, J. Zhang, N. Mamoulis, and Y. Tao, "Query Processing in Spatial Network Databases," *Proc. 29th Int'l Conf. Very Large Data Bases (VLDB),* pp. 802-813, 2003.

[21] M. Safar, "Group k-Nearest Neighbors Queries in Spatial Network Databases," *J. Geographical Systems,* vol. 10, no. 4, pp. 407-416, 2008.

[22] H. Samet, J. Sankaranarayanan, and H. Alborzi, "Scalable Network Distance Browsing in Spatial Databases," *Proc. ACM SIGMOD Int'l Conf. Management of Data,* pp. 43-54, 2008.

[23] M. Sharifzadeh, M.R. Kolahdouzan, and C. Shahabi, "The Optimal Sequenced Route Query," *VLDB J.,* vol. 17, no. 4, pp. 765-787, 2008.

[24] M. Sharifzadeh and C. Shahabi, "Processing Optimal Sequenced Route Queries Using Voronoi Diagrams," *GeoInformatica,* vol. 12, no. 4, pp. 411-433, 2008.

[25] R.W. White and D. Morris, "Investigating the Querying and Browsing Behavior of Advanced Search Engine Users," *Proc. 30th Ann. Int'l ACM SIGIR Conf. Research and Development in Information Retrieval,* pp. 255-262, 2007.

[26] M.L. Yiu, N. Mamoulis, and D. Papadias, "Aggregate Nearest Neighbor Queries in Road Networks," *IEEE Trans. Knowledge Data Eng.,* vol. 17, no. 6, pp. 820-833, June 2005.

[27] J.S. Yoo and S. Shekhar, "In-Route Nearest Neighbor Queries," *GeoInformatica,* vol. 9, no. 2, pp. 117-137, 2005.

[28] J. Zhang, M. Zhu, D. Papadias, Y. Tao, and D.L. Lee, "Location-Based Spatial Queries," *Proc. ACM SIGMOD Int'l Conf. Management of Data,* pp. 443-454, 2003.

Sarana Nutanong received the PhD degree in computer science from the University of Melbourne, Australia, in 2010. He is currently a postdoctoral research associate at the University of Maryland, College Park, Maryland. His research interests include spatial queries in road networks, spatio-temporal databases, and computational geometry.

Egemen Tanin received the PhD degree in computer science from the University of Maryland, College Park, Maryland, where he also held a postdoctoral research associate position from 2001 to 2003. He is a senior lecturer in the Department of Computer Science and Software Engineering, the University of Melbourne. His areas of research include spatial data management and database visualization.

Jie Shao received the BE degree in computer science from Southeast University, Nanjing, China, in 2004 and the PhD degree in computer science from The University of Queensland, Brisbane, Australia, in 2009. Currently, he is working as a research fellow in the Department of Computer Science and Software Engineering, The University of Melbourne, Australia. His research interests include multimedia information retrieval as well as spatial databases and their applications.

Rui Zhang received the bachelor's degree from Tsinghua University and the PhD degree from National University of Singapore. He is currently a senior lecturer in the Department of Computer Science and Software Engineering at the University of Melbourne, Australia. His research interest is data and information management in general, particularly in areas of indexing techniques, moving object management, web services, data streams and sequence databases.

Ramamohanarao (Rao) Kotagiri received the PhD degree from Monash University. He was awarded the Alexander von Humboldt Fellowship in 1983. He has been at the University Melbourne since 1980 and was appointed as a professor in computer science in 1989. He has held several senior positions including head of computer science and software engineering, head of the School of Electrical Engineering and Computer Science at the University of Melbourne and research director for the Cooperative Research Centre for Intelligent Decision Systems. He served on the editorial boards of the *Computer Journal*. At present, he is on the editorial boards of *Universal Computer Science, and Data Mining*, *IEEE Transactions on Knowledge and Data Engineering* and *VLDB (Very Large Data Bases) Journal*. He was the program cochair for VLDB, PAKDD, DASFAA, and DOOD conferences. He is a steering committee member of IEEE ICDM, PAKDD, and DASFAA. He received a Distinguished Contribution Award for Data Mining. He is a fellow of the Institute of Engineers Australia, the Australian Academy Technological Sciences and Engineering, and the Australian Academy of Science. He was awarded a Distinguished Contribution Award in 2009 by the Computing Research and Education Association of Australasia.