Multiple k Nearest Neighbor Query Processing in Spatial Network Databases

Xuegang Huang, Christian S. Jensen, and Simonas Šaltenis

Department of Computer Science, Aalborg University
Fredrik Bajers Vej 7E, DK-9220 Aalborg Øst, Denmark
{xghuang, csj, simas}@cs.aau.dk

Abstract. This paper concerns the efficient processing of multiple k nearest neighbor queries in a road-network setting. The assumed setting covers a range of scenarios such as the one where a large population of mobile service users that are constrained to a road network issue nearest-neighbor queries for points of interest that are accessible via the road network. Given multiple k nearest neighbor queries, the paper proposes progressive techniques that selectively cache query results in main memory and subsequently reuse these for query processing. The paper initially proposes techniques for the case where an upper bound on k is known a priori and then extends the techniques to the case where this is not so. Based on empirical studies with real-world data, the paper offers insight into the circumstances under which the different proposed techniques can be used with advantage for multiple k nearest neighbor query processing.

1 Introduction

A variety of location-based services for travelers such as tourists, visitors, and commuters are currently expected to be among the mobile services that have the highest likelihood of being used widely as the use of data services takes off.

An infrastructure is emerging that enables such services. In particular, vehicles are increasingly being equipped with general-purpose computing devices, e.g., in-board devices and aftermarket PDAs and dedicated navigation devices, and cellular data connections, e.g., GSM/GPRS and UMTS. Mobile users may thus request services from a central server, and these services will involve the processing of spatial queries, among which k nearest neighbor (kNN) queries are expected to be frequent.

This general scenario underlies a number of recent contributions to spatial query processing. In particular, it is reasonable to assume that the service users are constrained to a road network and that points of interest located in the road network are of interest to the services. Contributions exist that consider a variety of spatial queries in this setting, including range queries, closest-pair queries, distance joins, and also kNN queries. However, existing contributions focus on efficient means of answering a single query.

In contrast, it is reasonable to expect that the central server will at times receive many query requests, making it important to not simply consider the efficient processing of each query in isolation, but to process multiple queries efficiently, thus obtaining improved throughput. This paper does exactly that. The idea underlying multiple spatial query processing is to re-use cached results of recently computed, nearby queries

Y. Manolopoulos, J. Pokorný, and T. Sellis (Eds.): ADBIS 2006, LNCS 4152, pp. 266–281, 2006.

for computing a location-dependent query. The restriction of the mobile users and the points of interest to a road network contributes to making such re-use effective.

This paper thus considers the efficient processing of multiple kNN queries. More specifically, it presents a range of approaches for the main-memory caching and re-use of previously computed queries; and it reports on empirical studies of its proposals that utilize real-world road network and points of interest data. The caching approaches proposed are relatively easy to implement. Since it is also easy to switch from one approach to another, it is possible to combine the approaches so that the currently best approach is always utilized. The empirical studies suggest that the paper's proposals yield better performance than the existing single-query processing approach.

We believe that the contributions made by the paper are applicable to other kNN algorithms than the one considered, and we believe that they are applicable also to other types of spatial queries than kNN queries.

Query processing in the context of spatial networks as well as kNN query processing have recently attracted significant attention, and several papers are available that concern kNN and related queries for spatial networks [2,5,6,7,8,11]. One approach, the INE algorithm [11], uses variation of Dijkstra's algorithm for incremental network expansion, in that way computing a kNN query. In contrast, other approaches [2,5,6,8] pre-compute local distances to data objects or kNNs and store these on disk, so that subsequent kNN queries can be processed more efficiently. These approaches all consider the processing of queries one at a time, and they use disk-based structures. In contrast, our focus is on the efficient processing of multiple kNN queries by using main-memory caching strategies. This paper's proposal uses a modified INE algorithm.

Past proposals have utilized different storage structures for spatial networks. This paper adopts the data structures proposed along with the INE and Islands approaches [5,11], which are also similar to the CCAM [13] structure. Among the existing spatial network models [4,12], we adopt the link-node representation of a road network.

Within spatial databases, existing papers [9,10,14] discuss the processing of multiple queries by assuming that objects move in Euclidean space. Specifically, techniques [10] have been proposed for processing multiple range queries with the idea of ordering the queries so that "similar" queries are close and can be executed together. For continuously answering a collection of concurrent continuous kNN queries, the SEA-CNN approach [14] groups similar queries in a query table so that these continuous kNN queries are reduced to a spatial join between the objects and queries. The conceptual partitioning monitoring (CPM) algorithm [9] partitions the space around each query with a 2-dimensional grid and improves the nearest neighbor search on the grid by organizing the cells into conceptual rectangles for each query. In contrast, we consider the processing of multiple static k nearest neighbor queries in spatial networks. This functionality is novel and also essential for continuous kNN query processing in spatial networks where static kNN queries have to be computed several times during a single continuous query.

The paper is outlined as follows. Section 2 presents the background of this paper. Section 3 introduces the multiple query processing approaches and their extensions. The performance of these approaches is studied in Section 4. Finally, Section 5 summarizes the paper and offers directions for future research.

2 Background

In the prototypical usage scenario for this paper's contribution, a population of on-line users move in a road network (e.g., by foot, bicycle, bus, or car) while issuing requests to a central server for location-based services. The services involve kNN queries for points of interest (e.g., gas stations or attractions) that are located within the road network. The objective is now for the server to be able to process as many queries as possible. Terming the users *query points* and the points of interest *data points*, we proceed to consider the modeling of this scenario in more detail.

2.1 The Road Network Model

A *road network* is defined as a two-tuple $RN = (G, co\mathcal{E})$, where G is a directed, labeled graph and $co\mathcal{E}$ is a binary, so-called co-edge, relationship on edges. The graph G is itself a two-tuple (V, E), where V is a set of vertices and E is a set of edges. Vertices model intersections and starts and ends of roads. An edge e models the road in-between an ordered pair of vertices and is a three-tuple $e = (v_s, v_e, l)$, where $v_s, v_e \in V$ are, respectively, the start and end vertex of the edge. The edge can be traversed only from v_s to v_e. The element l captures the travel length of the edge. A pair of edges (e_i, e_j) belong to $co\mathcal{E}$, if and only if they represent the same bi-directional part of a road and a u-turn is allowed from e_i to e_j.

Next, a *location* loc in the road network is a two-tuple (e, pos) where e is the edge where the location is located and pos represents the length from the start vertex of the edge to loc. Then, a *data point* is modeled as a non-empty set of locations, i.e., $dp = \{loc_1, \cdots, loc_k\}$.

A *query point* qp is modeled as a two-tuple (e, pos) where e is the edge on which the query point is located and pos represents the length from the start vertex of the edge to qp. Given a query point and a value k, the kNN query returns k data points for which no other data points are closer to the query point in terms of road-network distance. The distance between a query point and a data point is the length of a shortest path between the query point and the location of the data point that is closest to the query point.

An edge with start and end vertices v_i and v_j is denoted by $e_{i,j}$. Figure 1 illustrates the concepts defined above, e.g., edge $e_{3,4} = (v_3, v_4, 6)$, data point $dp_1 = \{(e_{3,4}, 5), (e_{4,3}, 1)\}$, and query point $qp = (e_{8,9}, 2)$.

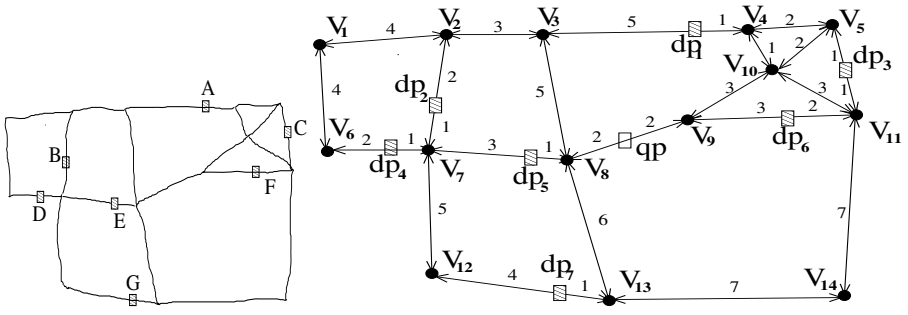


Fig. 1. Road Network Model

For the simplicity of our discussion, we assume that each edge in the road network has a corresponding co-edge connecting the two vertices in the opposite direction. Each data point then has two positions—one on each edge that models the road along which the data point is located. Note that in figures (as in Figure 1) we draw the two co-edges as one edge with two arrows.

2.2 INE Revisited

The INE algorithm is an adaptation of Dijkstra's shortest-path algorithm to use a disk-based network data structure [11]. It incrementally expands its search for data points through a network, starting at a query point. At each step, it reads the closest vertex w from a priority queue, Q_v, which stores yet-to-be-visited vertices in the order of their network distance from the query point. Then it puts all non-visited adjacent vertices of w into Q_v and inserts the data points found on the adjacent edges of w into a queue Q_{dp} that stores the data points found so far. Let d_k denote the network distance from the query point to the kth nearest neighbor in Q_{dp}. The search terminates when k nearest neighbors are found and the distance from the query point to the next vertex to be explored is larger than d_k. For the example road network in Figure 1, Figure 2 illustrates the steps of the INE algorithm for a 3NN query at $qp = (e_{8,9}, 2)$.

Step	Q_v	Q_{dp}	d_k
1	$\langle(v_8,2),(v_9,2)\rangle$	$\emptyset$	∞
2	$\langle(v_9,2),(v_7,6),(v_3,7),(v_{13},8)\rangle$	$\langle(dp_5,3)\rangle$	∞
3	$\langle(v_{10},5),(v_7,6),(v_3,7),(v_{11},7),(v_{13},8)\rangle$	$\langle(dp_5,3),(dp_6,5)\rangle$	∞
4	$\langle(v_4,6),(v_7,6),(v_3,7),(v_{11},7),(v_5,7),(v_{13},8)\rangle$	$\langle(dp_5,3),(dp_6,5)\rangle$	∞
5	$\langle(v_7,6),(v_3,7),(v_{11},7),(v_5,7),(v_{13},8)\rangle$	$\langle(dp_5,3),(dp_6,5),(dp_1,7)\rangle$	7
6	$\langle(v_3,7),(v_{11},7),(v_5,7),(v_{13},8),$ $(v_2,9),(v_6,9),(v_{12},11)\rangle$	$\langle(dp_5,3),(dp_6,5),(dp_1,7),$ $(dp_2,7),(dp_4,7)\rangle$	7

Fig. 2. Steps for 3NN Using the INE Algorithm

2.3 System Architecture

We assume a client-server architecture: mobile users issue requests that involve kNN queries from their mobile devices to a central server that perform the processing. If, during a short time span, more queries arrive than the server can process, they are queued. As answering a kNN query entails accessing a certain amount of road network data, only some of which can be cached in main memory, the focus of this paper is to minimize the number of disk accesses to the road network data needed for answering multiple queries. Queries are put in a queue based on their arrival order. The road network model and the points of interest are also managed by the server. In each iteration, the query processor takes one query request from the queue and processes it by accessing these data sets. We omit the description of the detailed structures used for the network model and points of interest, as we simply re-use those described for the INE and Islands approaches [5,11].

3 Multiple *k*NN Processing Algorithms

By caching results of previously answered kNN queries in main memory, it becomes possible for a new query to experience a reduction of accesses to disk-resident road network data if it is able to re-use cached data. We denote the conventional algorithm that simply processes the multiple queries as they arrive using the INE approach as $Conv\_kNN$. We proceed to introduce three approaches that improve the multiple kNN query processing when an upper-bound on k is known, and then we extend the algorithms to the general case.

3.1 The Case of Known Upper Bound on *k*

We assume an upper bound $k_{\max}$ on the k in the multiple kNN queries, i.e., $k \leq k_{\max}$. Such a bound may be realistic in real-world applications, as it can either be pre-defined by LBS vendors or be obtained by observing historical records.

Basic Observation

Lemma 1. Let qp be a query point, v a network vertex, and dp a data point. If dp is one of the k nearest neighbors of qp and the shortest path from qp to dp passes through v, then dp is also one of the k nearest neighbor data points of v.

Based on this lemma, during the kNN expansion process from a query point qp, if a network vertex v is visited and the k nearest neighbor data points of v are already known, the expansion process reuses these k nearest neighbors of v and avoids visiting adjacent vertices of v. This is possible because the INE algorithm guarantees that when v is visited (removed from the queue of vertices), the shortest path from qp to v has already been found. This, combined with Lemma 1, guarantees that all qp's k nearest neighbors, which have the shortest paths from qp passing through v, can be found among the k nearest neighbor data points of v.

With this observation, if we cache a certain amount of network vertices together with their k nearest data points, a newly-started kNN expansion process will be able to re-use the cached data and save computation.

We extend the INE algorithm with the capability of using the cached data. The extended algorithm, INE^*, takes three parameters: the query point qp, the value k, and a list L of cached results. Entries in the list L have the form (v, QP^v), where v is a vertex and QP^v is the set of the k nearest data points of v (including corresponding distance values). Similar to the INE approach, during the network expansion process, the INE^* algorithm uses two priority queues, Q_{dp} and Q_v, to record, respectively, data points and vertices together with their distance to the query point, denoted as $d(qp, dp)$ and $d(qp, v)$. Both queues sort elements by the distance value and do not allow duplicate data points or vertices. The size of Q_{dp} is limited to k elements. We introduce *update* and *deque* operations for the two queues. The *update*(dp/v, *dist*) operation inserts a new data point or vertex and the corresponding distance into the queue. If this data point or vertex is already in the queue then, if *dist* is smaller than the distance stored in the queue, the distance value in the queue is updated to *dist*. The *deque* operation removes a vertex with the smallest distance and returns it. The pseudo code is listed next. Queues Q_v and Q_{dp} are assumed to be empty initially.

(1) **procedure** $INE^*(qp, k, L)$
(2) **for each** data point *dp* on edge *qp.e*: Q_{dp}.*update*$(dp, d(qp, dp))$
(3) $Q_v.update(qp.e.v_s, d(qp, qp.e.v_s)), Q_v.update(qp.e.v_e, d(qp, qp.e.v_e))$
(4) **if** $\exists a$ such that $(a, qp.e) \in co\mathcal{E}$,do lines (2)–(3) assuming $qp = (a, a.l - qp.pos)$
(5) Let dp_k denote the k-th element in Q_{dp}, or $dp_k = \perp$ if there is no such element
(6) $d_k \leftarrow d(qp, dp_k)$ // $d_k \leftarrow \infty$ if $dp_k = \perp$
(7) $v_x \leftarrow Q_v$.*deque*, mark v_x visited
(8) **while** $d(qp, v_x) < d_k \wedge Q_v \neq \emptyset$
(9) **if** $(v_x, QP^{v_x}) \in L$
(10) **for each** $dp \in QP^{v_x}$: Q_{dp}.*update*$(dp, d(qp, v_x) + d(v_x, dp))$
(11) **else**
(12) **for each** non-visited adjacent vertex v_y of v_x
(13) **for each** *dp* on edge $e_{x,y}$ (and edge $e_{y,x}$ if $(e_{x,y}, e_{y,x}) \in co\mathcal{E}$)
(14) Q_{dp}.*update*(*dp*, $d(qp, v_x) + d(v_x, dp)$)
(15) Q_v.*update*$(v_y, d(qp, v_x) + e_{x,y}.l)$
(16) $d_k \leftarrow d(qp, dp_k)$
(17) $v_x \leftarrow Q_v$.*deque*, mark v_x visited
(18) **return** Q_{dp}

During the INE^* expansion process, whenever a vertex v_x in the list L is visited, the algorithm updates the queue Q_{dp} with the kNNs of v_x (line 10) and proceeds to visit the next vertex in the queue Q_v (line 17). It can be observed that the algorithm still works if the list L keeps more than k nearest neighbors to corresponding query points. Then line 10 only uses the first k data points of v_x. With this algorithm as a basis, we introduce three approaches for multiple k nearest neighbor query processing.

The Sharing Approach. A basic approach to improving the efficiency of multiple query processing is to re-use the results of finished kNN queries for new queries. Since these finished query points can be treated as extra vertices on the road network, Lemma 1 applies, and the INE^* algorithm can be used. To control the size of the list of cached query results (list L), we define a threshold $\mathcal{D}$ and add this threshold as an additional parameter to the INE^* algorithm. For a query started at qp, if a cached query point qp' is discovered in the network expansion process within a network distance $\mathcal{D}$ from qp, the result of the query at qp is not saved in L. Otherwise, it is saved in the list for future queries.

Assuming a sequence of queries $\mathcal{S} = \langle \ldots, (qp_i, k_i), \ldots \rangle$, where qp_i is a query point and k_i is the number of nearest neighbors ($0 < k_i \leq k_{\max}$), we describe the sharing approach in the following.

Approach 1. (The $S\_kNN(\mathcal{S}, \mathcal{D}, k_{\max})$ algorithm)

1. Retrieve query request (qp_i, k_i) from $\mathcal{S}$
2. Execute $INE^*(qp_i, k_i, L, \mathcal{D})$; in the expansion process, if k_i neighbors are found within $\mathcal{D}$ while no cached query points are discovered, continue the expansion to distance range $\mathcal{D}$ or until a cached query point is reached; If there are no cached query points found within $\mathcal{D}$, continue the expansion until $k_{\max}$ neighbors are found and save the query result (qp_i, QP^{qp_i}) in L
3. Go to step 1 until $\mathcal{S} = \emptyset$ □

Step 2 of the approach guarantees that no two cached queries are closer to each other than $\mathcal{D}$ and that all of the cached results contain $k_{\max}$ neighbors. If k nearest neighbors are found within the distance threshold $\mathcal{D}$ from the query, the algorithm continues the expansion to distance $\mathcal{D}$ to check if the query has to be cached.

An alternative policy is to cache a query point if its k nearest neighbors are found within $\mathcal{D}$ and no other cached query points are reached in the process. With this policy, parts of the road network with a high density of data points will obtain many cached queries. This, in turn, may result in queries from other areas of the road network being purged from the cache due to its limited size. In this way, areas dense with data points are favored in the cache, and this may not be desirable because, even without caching, queries run fast in these areas due to small expansion ranges. Thus, we choose to enforce the threshold $\mathcal{D}$ strictly, which results in a uniform distribution of cached queries in the road network.

The Clustering Approach. Intuitively, if a number of queries are clustered in a small area of the road network, most of them will benefit from queries cached near the cluster. In the following, we explore a approach that finds the clusters of queries in order to obtain maximum reuse of cached query results within the clusters.

We divide the road network into "sub-networks" generated by the clusters of query points (details will follow). Consider Figure 3. The network inside the big rectangle R is a sub-network of the example road network in Figure 1. We denote this sub-network R. A network vertex belongs to R if, based on coordinates of this vertex, it is inside the rectangle R. We divide all vertices belonging to R into two types. First, those vertices whose adjacent vertices also belong to R are called *internal vertices* of R. Second, those vertices that have at least one adjacent vertices not belonging to R are defined as *border vertices*. In Figure 3, vertex v_{10} is an internal vertex while vertices v_4, v_9, and v_{11} are border vertices. A network edge belongs to R if both its vertices belong to R, e.g., edges $e_{4,5}$ and $e_{9,11}$ belong to R in Figure 3. A data point or a query point belongs to a sub-network R if its edge belongs to R. As shown in Figure 3, data point dp_6 and query point qp_2 belong to R while dp_1 and qp_1 do not.

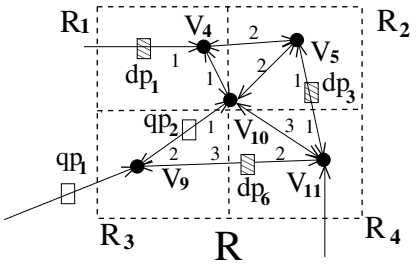


Fig. 3. The Clustering Approach

The clustering approach is based on the following observation. In Figure 3, suppose a 3NN query is issued from query point qp_2 in R. We can answer the query in two steps. First, we run the INE^* algorithm to find 3NNs to all border vertices of R: v_4, v_9, v_{11}. Second, we run the INE^* at qp_2, but during the incremental expansion process, when a border vertex is visited, we treat it as a cached query—its corresponding 3NNs (computed in the first step) are added into queue Q_{dp} and the expansion process does not proceed to the adjacent vertices. Since 3NNs of all border vertices are pre-computed, the network expansion process is constrained inside R. The 3NNs of qp_2 are data points found either by the expansion process inside R or by reading nearest neighbors of border vertices. Based on Lemma 1, the result of such a two-step execution is correct.

Although this procedure restricts the expansion scope of a kNN query to a sub-network, it is expensive to answer a single kNN query in such a way due to the cost of pre-computing kNNs of border vertices. However, since the pre-computed data can be used for all the query points inside the same sub-network, sharing of the pre-computed border vertices may be beneficial if a substantial amount of queries are running in the same sub-network. In addition, we "pre-compute" the border vertices in a lazy fashion—a kNN query on a border vertex is run and the result is cached only when we first encounter this vertex during the processing of some query.

To generate sub-networks, we assume that a spatio-temporal histogram $\mathcal{H}$ is available. It is a uniform two-dimensional $m \times m$ grid covering the MBR (Minimum Bounding Rectangle) of the whole road network. Each histogram cell records the number of query points located in this cell in a short history. We use the $DBSCAN$ algorithm [3] to cluster the histogram cells based on the recorded numbers of query points. A cluster's ID is then recorded with each cell of the cluster. Cells that are not assigned to any cluster by $DBSCAN$ are assigned to the "cluster" of outliers.

The modified INE^* algorithm gets $\mathcal{H}$ as an extra parameter and uses the cluster IDs of grid cells to determine border vertices in the network expansion process. When examining a vertex, the algorithm uses the coordinates of the vertex to find its histogram cell and the corresponding cluster ID. By comparing the cluster IDs of the vertex and all its adjacent vertices, the algorithm determines if the vertex is a border vertex. For example, suppose the four small rectangles in Figure 3 are histogram cells and are assigned the same cluster ID while their neighboring rectangles (not shown in the figure) have different cluster IDs. Vertex v_9 is a border vertex because it is inside the cluster while one of its adjacent vertex is in a cell of a different cluster.

The clustering approaches takes the following parameters: a sequence of queries $\mathcal{S}$, the histogram $\mathcal{H} = \{c_1, c_2, \ldots, c_m\}$, the upper-bound $k_{\max}$, and the $DBSCAN$ parameters Eps and $MinPts$ [3]. Briefly, Eps defines a distance scope for searching neighborhood points and $MinPts$ defines the minimum number of points in a neighborhood to a "center" point. We proceed to consider the clustering approach in more detail.

Approach 2. (The $C\_kNN(\mathcal{S}, \mathcal{H}, k_{\max}, Eps, MinPts)$ algorithm)

1. Execute $DBSCAN(\mathcal{H}, Eps, MinPts)$ saving cluster IDs with each cell in $\mathcal{H}$
2. Retrieve (qp_i, k_i) from $\mathcal{S}$
3. Execute $INE^*(qp_i, k_i, L, \mathcal{H})$; in the expansion process, if a border vertex v is visited, do not consider its adjacent vertices (lines 12–15 in INE^*). If v is in L, update Q_{dp} with k_iNNs of v (line 10). If v is not in L, execute $INE^*(v, k_{\max}, L)$, placing the result (v, QP^v) into L and update Q_{dp}.
4. Go to step 2 until $\mathcal{S} = \emptyset$ □

As discussed, the cached list L, which is used to record border vertices of clusters and their kNNs, is populated in a lazy fashion. When enough border vertices of a cluster are computed, network expansions starting inside the cluster will have a reduced scope.

For an example of the running of this algorithm, consider the sub-network covered by rectangle R in Figure 3 as a sub-network of the whole network in Figure 1. Assume that a number of queries were already processed in this sub-network, so that 3NNs to the border vertices are computed (shown in Figure 4(a)). Then, Figure 4(b) demonstrates the running steps of $INE^*(qp_2, 3, L, \mathcal{H})$ at $qp_2 = (e_{9,10}, 2)$.

Border Vertex	3NNs
v_4	$\langle(dp_1, 1), (dp_3, 3), (dp_6, 6)\rangle$
v_9	$\langle(dp_6, 3), (dp_1, 5), (dp_5, 5)\rangle$
v_{11}	$\langle(dp_3, 1), (dp_6, 2), (dp_1, 5)\rangle$

(a) List L

Step	Q_v	Q_{dp}	d_k
1	$\langle(v_{10}, 1), (v_9, 2)\rangle$	$\emptyset$	∞
2	$\langle(v_9, 2), (v_4, 2), (v_5, 3), (v_{11}, 4)\rangle$	$\emptyset$	∞
3	$\langle(v_4, 2), (v_5, 3), (v_{11}, 4)\rangle$	$\langle(dp_6, 5), (dp_1, 7), (dp_5, 7)\rangle$	7
4	$\langle(v_5, 3), (v_{11}, 4)\rangle$	$\langle(dp_1, 3), (dp_3, 5), (dp_6, 5)\rangle$	5
5	$\langle(v_{11}, 4)\rangle$	$\langle(dp_1, 3), (dp_3, 4), (dp_6, 5)\rangle$	5
6	$\emptyset$	$\langle(dp_1, 3), (dp_3, 4), (dp_6, 5)\rangle$	5

(b) Steps for 3NN from qp_2

Fig. 4. Running Example of INE^* in C_kNN

The Combined Approach. In an attempt to combine the benefits of the sharing and clustering approaches, we combine step 3 of the C_kNN algorithm with step 2 of the S_kNN algorithm. The combined approach takes six parameters: the sequence of queries $\mathcal{S}$, the histogram $\mathcal{H}$, the upper bound $k_{\max}$, clustering parameters Eps, $MinPts$, and the threshold $\mathcal{D}$. We describe the approach in the following.

Approach 3. (The $SC\_kNN(\mathcal{S}, \mathcal{H}, k_{\max}, Eps, MinPts, \mathcal{D})$ algorithm)

Execute $C\_kNN(\mathcal{S}, \mathcal{H}, k_{\max}, Eps, MinPts)$ with the following modifications: In step 3, execute $INE^*(qp_i, k_i, L, \mathcal{H})$; in the expansion process, if a border vertex v is visited, do not consider its adjacent vertices (lines 12–15 in INE^*). If v is in L, update Q_{dp} with the k_iNNs of v (line 10). If v is not in L, run $INE^*(v, k_{\max}, L)$, put the result (v, QP^v) into L, and update Q_{dp}. If k_i neighbors are found within $\mathcal{D}$ while no cached query points are discovered, continue the expansion to distance range $\mathcal{D}$ or until a cached query point is reached; if there are no cached query points found within $\mathcal{D}$, continue the expansion until $k_{\max}$ neighbors are found and save the query result (qp_i, QP^{qp_i}) in L. □

Here, list L contains two types of cached results—results of previous queries and for border vertices. We assign equal weight to both types and use LRU cache-replacement.

3.2 The Case of Unknown Upper Bound on *k*

As described, the S_kNN, C_kNN, and SC_kNN algorithms assume a have fixed upper-bound on k. Such an assumption, although is applicable in real LBS applications, limits the flexibility of these applications. Thus, we proceed to extend the algorithms to process queries with arbitrary k values.

To see how the S_kNN algorithm can be extended, suppose a k_1NN query at query point qp_1 in Figure 5 is processed and cached. When the k_2NN query at qp_2 visits qp_1, if $k_2 \leq k_1$, based on Lemma 1, the network expansion process can update the result with the first k_2 nearest data points of qp_1 and stop visiting neighbor vertices of qp_1. If

$k_2 > k_1$, the network expansion can also use the k_1 nearest data points of qp_1, but it has to continue visiting adjacent vertices of qp_1. The bigger the sizes (k's) of the cached query results, the better such a strategy works.

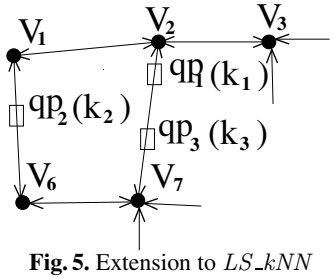


Fig. 5. Extension to LS_kNN

To achieve high k's of the cached query results, we exchange a cached query point with a new query point with a higher k, whenever such a new query is issued on the same edge or co-edge. For example in Figure 5, when another k_3NN query at qp_3 is issued and qp_3 is on the same edge as qp_1, after processing of qp_3, if $k_3 > k_1$, we replace the cached qp_1 with qp_3 and corresponding nearest neighbors. This way, the sizes (k's) of the cached query results is increased lazily, as queries with high k's arrive.

We summarize the "lazy-update" sharing approach in the following. The parameters for the algorithm are the same as for S_kNN, except for the upper bound of k.

Approach 4. (The $LS\_kNN(\mathcal{S}, \mathcal{D})$ algorithm)

Execute the S_kNN algorithm with the following modifications. In step 2, in the expansion process of $INE^*(qp_i, k_i, L, \mathcal{D})$, when a cached query point is encountered and L is updated with its nearest neighbors, if its k value is smaller than k_i, continue visiting its adjacent vertices. Before step 3, if there is another query point on the same edge as qp_i with a smaller k value than k_i, replace that query point and its corresponding nearest neighbors with qp_i and its nearest neighbors. □

With this "lazy-update" strategy, the LS_kNN algorithm is able to process multiple kNN queries without setting the upper bound of k. Notice that the efficiency of the strategy largely depends on the distribution of k values in the query stream. The worst case for the algorithm is when k values are small at the beginning of a query stream and increase with time. Also notice that, by replacing cached query points with new ones on the same edge, the enforcement of the precise threshold $\mathcal{D}$ is compromised.

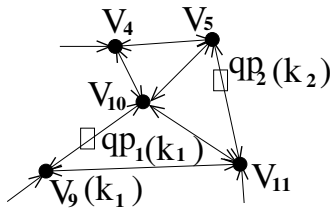


Fig. 6. Extension to LC_kNN

We can also apply the lazy-update strategy to the C_kNN algorithm. As shown in Figure 6, let the border vertices of the sub-network be v_4, v_5, v_9, and v_{11}. Suppose also that after processing the k_1NN query at qp_1, border vertex v_9 is cached with k_1 nearest neighbors. Then, when a k_2NN query at qp_2 visits the border vertex v_9, if $k_2 \leq k_1$, the expansion process updates the query result with the first k_2NNs of v_9 and avoids visiting its adjacent vertices. If $k_2 > k_1$, the network expansion is paused and a new k_2NN query is fired at v_9 to find k_2 nearest neighbors. Then the query uses these NNs of v_9 to update the query result and continues expanding in other directions. The cached k_1NNs of v_9 are replaced with its k_2NNs.

The pseudo code for this lazy-update clustering approach follows. It uses the same parameters as the C_kNN algorithm, except from the upper bound of k.

Approach 5. (The $LC\_kNN(\mathcal{S}, \mathcal{H}, Eps, MinPts)$ algorithm)
Execute the C_kNN algorithm with the following modification. In step 3, in the expansion process of $INE^*(qp_i, k_i, L, \mathcal{H})$, when a border vertex v is visited, if v is in L and has no less than k_i cached NNs, update Q_{dp} with the k_iNNs of v. If v is not in L or it has less than k_i cached NNs, run $INE^*(v, k_i, L)$, place the result (v, QP^v) in L, and update Q_{dp}. □

We can also extend the SC_kNN algorithm by applying the above-described strategies. We omit the presentation of the "Lazy-Combined Approach" (denoted as **Approach 6**) and denote the algorithm as the LSC_kNN algorithm. It has the same parameters as the SC_kNN algorithm, but is able to handle multiple nearest neighbor queries with arbitrary k values.

3.3 Discussion

As pointed out in the coverage of the VN3 and Island approaches [5,8], for an online-processing system, it is necessary to consider updates to the road network as well as points of interest during query processing. For the algorithms proposed in this paper, updates to both the network and data points will cause the cached list L to be truncated and re-filled by new queries. In addition, since the Islands approach uses a similar network expansion algorithm as the INE algorithm, the approaches proposed here can be directly applied with the Islands approach. It will be an interesting direction to consider how to accommodate updates to the network and points of interest data while, at the same time, improve the efficiency of processing multiple queries.

As we have proposed a total of 6 approaches, we believe that, since the different approaches may perform best in different situations, it is possible to design query execution strategies that, based on given situations, automatically switch among these approaches to always achieve the best performance. The switching among the six approaches is straightforward since one only needs to replace the network expansion strategy in the INE^* algorithm. In the next section, we focus on experimentally exploring the settings for which each of the approaches excels.

4 Evaluation

We use two data sets for examining the performance properties of the caching approaches. The first consists of a real-world road network and associated points of interest for Aalborg (AAL), Denmark, containing $11,300$ vertices, $13,375$ bi-directional edges, and 279 data points. The second data set is a representation of the road network of San Francisco (SF) [1]. It contains $175,343$ vertices as well as $223,140$ bi-directional edges. The road network and points of interest data are arranged into disk pages based on the data structures described for the INE and Islands approaches [5,11]. We set the page size to 4k and use an LRU buffer for caching the disk pages read by the algorithms. While the $Conv\_kNN$ algorithm uses the whole main-memory buffer for the LRU buffer of disk pages, the algorithms proposed in the paper also use an in-memory list L that occupies part of the main-memory buffer. We also apply the LRU strategy to L. The total size of the buffer is 15% of the network data. The AAL and SF datasets

contain 129 and $4,023$ pages. We study the performance of these approaches in terms of the average disk accesses. The approaches are implemented in C++ (the $DBSCAN$ algorithm is based on the source code kindly provided to us by its authors [3]).

Values for parameters used in the experiments are listed in Figure 7 (the values in bold are defaults). Briefly, they are the number of query points, the range of k, the size of the cached list L, the distance threshold $\mathcal{D}$, the number of histogram cells, and parameters Eps and $MinPts$ for the clustering algorithm. We define the maximum Euclidean distance between any two vertices of the road network as $D_{\max}$. The distance threshold $\mathcal{D}$ is represented as a fraction of $D_{\max}$. The histogram is a uniform $m \times m$ grid exactly covering the MBR of the whole road network. We define C_l as the length of the diagonal of a histogram cell, and C_{ave} as the average number of query points inside an "occupied" cell, i.e., a cell containing at least one query point. The parameter Eps is represented as a function of C_l, and $MinPts$ as a function of C_{ave}.

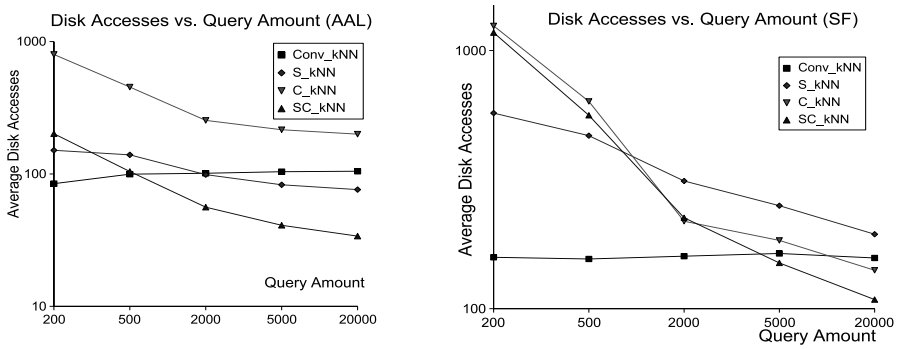
Query Points	$200, 500, 2000, \mathbf{5000}, 20000$
Range of k	$[1,5], [1,10], [\mathbf{1}, \mathbf{20}], [1,50], [1,80]$
Size of List L	$0.1, 0.15, \mathbf{0.2}, 0.25, 0.3, 0.35, 0.4$ of the buffer size
Threshold $\mathcal{D}$	$0.001, 0.005, \mathbf{0.01}, 0.05, 0.1$ of $D_{\max}$
Histogram Cells	$5 \times 5, 8 \times 8, \mathbf{10 \times 10}, 15 \times 15, 20 \times 20$
Eps	$\mathbf{2 * C_l}$
$MinPts$	$\mathbf{0.5 * C_{ave}}$

Fig. 7. Parameter Values

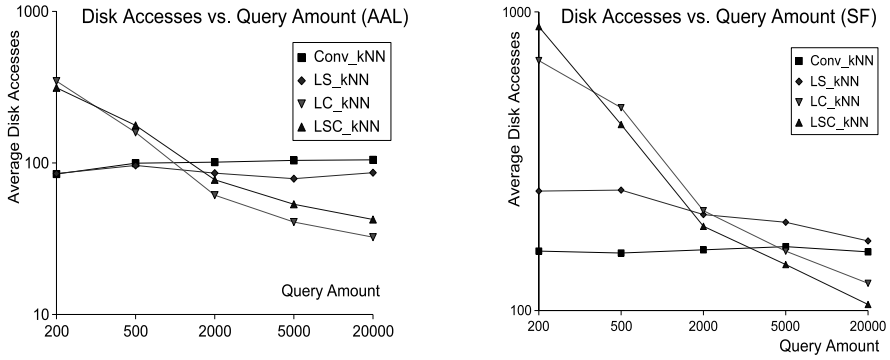
We use the real data points in the AAL data set and introduce synthetic data points for the SF data set in our evaluations. The synthetic data points are generated randomly at a density of 0.1%, where the density is defined as the number of data points versus the number of bi-directional edges in the network. The query points and k values (within a given range) are generated randomly.

In the experiments, we first explore the differences among the **sharing**, **clustering**, and **combined** approaches. As described, updates to network data cause the cached list L to be invalidated for all approaches. Depending on the frequency of such updates, the average number of queries issued in-between two resettings of the cached list L may vary. In the first set of experiments, we explore the average query performance for varying amount of queries. The average number of disk accesses per query is measured, and the experiments are run on both the AAL and SF data sets, for both cases with and without a known upper bound on k. The parts of the curves to the right in Figure 8 describe the performance when updates are infrequent, while the parts of the curves towards the left represent the performance when updates are increasingly frequent.

For the **clustering** approach, at the beginning of each experiment, all the query points to be executed are clustered. Assuming that the query distribution does not change with time, the resulting sub-networks should be similar to the sub-networks generated by clustering a history of past queries as described in Section 3. Figure 8 shows that the **sharing** approach is competitive with the conventional algorithm in the AAL network, but has worse performance in the SF network. The results also demonstrate that the **clustering** and the **combined** approaches have high costs for very small amounts of query points. Thus, when the cached list L is invalidated too often, which happens when updates occur, the approaches are worse than $Conv\_kNN$.



(a) Average Performance with Upper Bound of k

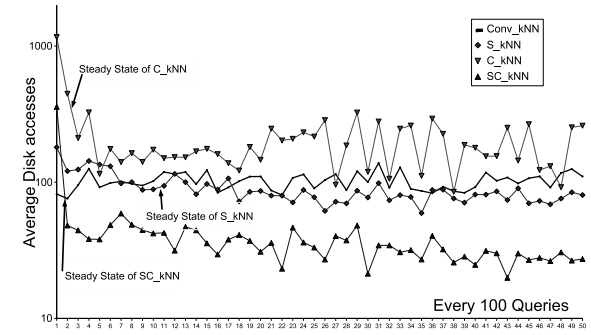


(b) Average Performance without Upper Bound of k

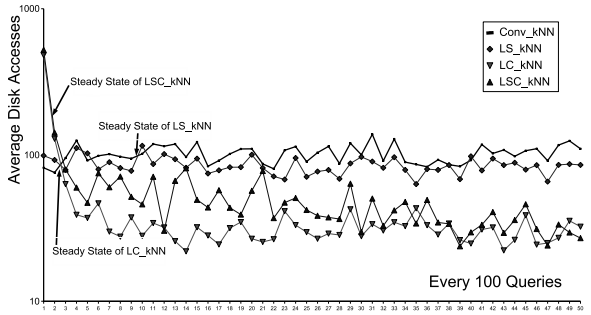
Fig. 8. Accumulated Query Performance

To study in detail how the cached data influence query efficiency, we perform $5,000$ queries (on the AAL data set) and measure the average disk accesses for every 100 queries. We define the "steady state" for the cached list L as the first time it becomes full. As illustrated in Figure 9, the performances of the **sharing** approaches are very close to that of the conventional algorithm, but exhibit slightly better performance than $Conv\_kNN$ after the "steady state."

The **clustering** and **combined** approaches both show substantially improved query performance after the steady state. An interesting observation is that the **clustering** algorithm with an upper bound of k (C_kNN) has the worst performance of all (see Figure 9(a)), while the variant without an upper bound of k, LC_kNN, is the best one (see Figure 9(b)). This is because the upper bound of k in the first case is used by kNN queries at border vertices. Depending on the value of $k_{\max}$, each such query has a substantial cost and the corresponding cached result occupies substantial space in the list L. On the other hand, the LC_kNN algorithm incurs smaller cost for the queries at border vertices and uses less caching space to save the results of these queries, which, in turn, enables more items to be cached in L.



(a) Disk Accesses with Upper Bound of k



(b) Disk Accesses without Upper Bound of k

Fig. 9. Evolution of Performance on AAL Network

Figures 9(b) and 8(b) show that the LC_kNN algorithm is slightly better than LSC_kNN for the AAL data set, while the same experiment on the SF data set shows that LSC_kNN outperforms LC_kNN. To further study the differences between these two approaches, experiments were performed varying other parameters: the size of the cached list L, the amount of cells in a histogram of queries, and the distance threshold $\mathcal{D}$ used by the LSC_kNN approach. Figure 10 shows the results for the AAL data set. It can be observed that the LSC_kNN algorithm performs better than the LC_kNN algorithm when the cache size is small, but it is outperformed by LC_kNN when the cache size grows. With more histogram cells, the LC_kNN algorithm seems to get worse and worse as the number of border vertices becomes too large compared to the given cache size. As expected, when the distance threshold $\mathcal{D}$ increases, less and less results of queries from non-border vertices are saved in the cache, and the performance of LSC_kNN becomes closer to LC_kNN. The difference between the LC_kNN and LSC_kNN algorithms is also affected by the network topology, the density and distribution of data

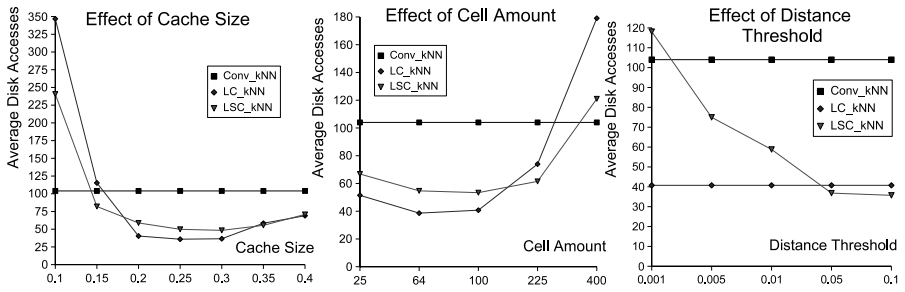


Fig. 10. Comparison of LC_kNN and LSC_kNN on Other Parameters (AAL)

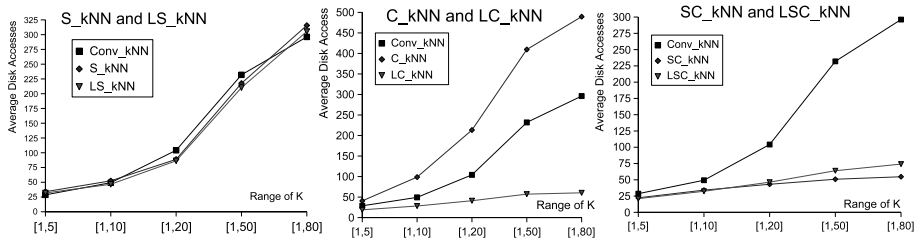


Fig. 11. Comparison of Approaches With or Without $k_{\max}$ (AAL)

points (the AAL data set includes real data points with a density of 2% and the SF data set has synthetic, uniformly distributed data points with a density of 0.1%), as well as the effect of the clustering functions.

Based on the described experiments, we conclude that with a tight upper bound on k that is not far from the average k value of the queries, the **combined** approach is the best suited approach. For the case where there is no such upper bound, both the **clustering** and the **combined** approaches have similar performance. The **clustering** approach may then be preferable because it is simpler than the **combined** approach.

To explore further the difference among the approaches when the upper bound of k is fixed or not, we execute $5,000$ queries for different ranges of k values. The parameter $k_{\max}$ is used in the S_kNN, C_kNN, $and SC\_kNN$ algorithms, while the "lazy" variants of these algorithms use actual k values as described in Section 3. As shown in Figure 11, the performances of the S_kNN and LS_kNN algorithms are quite close even with a very big upper bound of k. The **combined** approaches exhibit similar behavior. In contrast, the difference between the performances of C_kNN and LC_kNN is substantial. Comparing the performances of LC_kNN and SC_kNN, we conclude that the "lazy" **clustering** approach (LC_kNN) is the most suitable, independently of whether the upper bound of k is known or not.

Experiments were also performed to check how the performance of these algorithms is influenced by other parameters, i.e., density of data points and the clustering parameters Eps and $MinPts$. The results of these experiments, not covered in detail here, are quite consistent to those reported and thus provide a further validation of our findings.

Summarizing the performance evaluation, we can conclude that when the amount of successive queries between adjacent updates in a workload exceeds one thousand, the proposed approaches have better performance than the conventional approach, which uses the main-memory buffer solely as a disk-page buffer. Next, the "lazy" **clustering** approach (LC_kNN) is the most competitive of the proposed approaches under a broad variety of settings.

5 Summary and Research Directions

With focus on the use of main-memory caching strategies for improving the efficiency of multiple k nearest neighbor query processing, this paper presents a total of six caching algorithms. The paper first presents three basic approaches that assume that

an upper bound on k is known a priori. Then it extends these approaches to contend with the general case where the upper bound is unknown.

Empirical performance studies demonstrate that the algorithms excel over the conventional algorithm in a variety of circumstances. The algorithms termed the "lazy" **clustering** approach is the best in most settings. In addition, these algorithms are easy to implement and can be used in combination to achieve multiple k nearest neighbor query processing that outperforms existing proposals.

Future work can be explored in several directions. First, as discussed in the paper, it is relevant to consider updates to the network as well as the points of interest when processing multiple queries. Second, it is of interest to conduct a theoretical analysis of the relationships among parameters such as the cache size, the range of k, the query throughput, the data point density, and the performance of multiple queries. Third, it is of interest to investigate approaches that off-load the server side by delegating processing to the mobile devices.

Acknowledgments. C. S. Jensen is also an adjunct professor in Department of Technology, Agder University College, Norway.

References

1. T. Brinkhoff. Network-based Generator of Moving Objects. http://www.fh-oldenburg.de/iapg /personen/brinkhof/generator/
2. H. -J. Cho, C. -W. Chung. An Efficient and Scalable Approach to CNN Queries in a Road Network. In *Proc. VLDB,* pp. 865–876, 2005.
3. M. Ester, H. P. Kriegel, J. Sander, X. Xu. A Density-Based Algorithm for Discovering Clusters in Large Spatial Databases with Noise. In *Proc. KDD,* pp. 226–231, 1996.
4. R. H. Güting, V. T. de Almeida, and Z. Ding. Modeling and Querying Moving Objects in Networks. In *VLDB J.,* 2006, to appear.
5. X. Huang, C. S. Jensen, S. Šaltenis. The Islands Approach to Nearest Neighbor Querying in Spatial Networks. In *Proc. SSTD,* pp. 73–90, 2005.
6. H. Hu, D. L. Lee, J. Xu. Fast Nearest Neighbor Search on Road Networks. In *Proc. EDBT,* pp. 186–203, 2006.
7. C. S. Jensen, J. Kolář, T. B. Pedersen, I. Timko. Nearest Neighbor Queries in Road Networks. In *Proc. ACMGIS,* pp. 1–8, 2003.
8. M. Kolahdouzan, C. Shahabi. Voronoi-Based Nearest Neighbor Search for Spatial Network Databases. In *Proc. VLDB*, pp. 840–851, 2004.
9. K. Mouratidis, M. Hadjieleftheriou, D. Papadias. Conceptual Partitioning: An Efficient Method for Continuous Nearest Neighbor Monitoring. In *Proc. SIGMOD,* pp. 634–645, 2005.
10. A. Papadopoulos, Y. Manolopoulos. Multiple Range Query Optimization in Spatial Databases. In *Proc. ADBIS,* pp. 71–82, 1998.
11. D. Papadias, J. Zhang, N. Mamoulis, Y. Tao. Query Processing in Spatial Network Databases. In *Proc. VLDB,* pp. 802–813, 2003.
12. L. Speičys, C. S. Jensen, A. Kligys. Computational Data Modeling for Network Constrained Moving Objects. In *Proc. ACMGIS,* pp. 118–125, 2003.
13. S. Shekhar, D. Liu. CCAM: A Connectivity-Clustered Access Method for Networks and Network Computations. In *IEEE TKDE,* 19(1): 102–119, 1997.
14. X. Xiong, M. F. Mokbel, W. G. Aref. SEA-CNN: Scalable Processing of Continuous K-Nearest Neighbor Queries in Spatio-Temporal Databases. In *Proc. ICDE,* pp. 643–654, 2005.