Monitoring Heterogeneous Nearest Neighbors for Moving Objects Considering Location-Independent Attributes

Yu-Chi Su[1], Yi-Hung Wu[2], and Arbee L. P. Chen[3]

[1] Department of Computer Science, National Tsing Hua University, Taiwan, R.O.C.
2 Department of Information and Computer Engineering, Chung Yuan Christian University, Taiwan, R.O.C.
[3] Department of Computer Science, National Chengchi University, Taiwan, R.O.C.
alpchen@cs.nccu.edu.tw

Abstract. In some applications, data may possess both location-dependent and location-independent attributes. For example, in a job database, each job can be associated with both location-dependent attributes, e.g., the location of the work place, and location-independent ones, e.g., the salary. A person who uses this database to find a job may prefer not only a shorter distance between his/her house and the work place but also a higher salary. Therefore, a query with both concepts of "shorter distance" and "higher salary" should be considered to meet the user's needs. We call it the heterogeneous k-nearest neighbor (HkNN) query in distinction from the traditional k-nearest neighbor (kNN) query in the spatial domain, which only concerns location-dependent attributes. To our knowledge, this paper is the first work proposing a generic framework for solving the HkNN query. We propose an efficient approach based on the bounding property for the HkNN query evaluation. Furthermore, we provide an update mechanism for continuously monitoring the HkNN queries in a dynamic environment. Experimental results verify that the proposed framework is both efficient and scalable.

1. Introduction

The continuous k-nearest neighbor query processing over moving objects, which aims at retrieving the *k* objects closest to a query point, has been studied for several years [2][3][4][5][7][8][9]. However, in many applications, a moving object may also have attributes that are irrelevant to its location. The following are two examples drawn from different applications. Example 1 illustrates a scenario over static objects, while Example 2 is for moving objects in high dimensions.

Example 1. A person wants to buy shoes of a special brand. He/she plans to visit the best *k* shoe stores in order of the shoes' price and the expense for traveling to the store. Here we assume that the traveling expense for one distance unit (in kilometer) is 100 (in dollar) and the traveling expense equals the distance unit multiplied by the cost per distance unit. The total cost for buying new shoes at a store can be formulated as: Total cost = Price ($) + Traveling expense ($). The person's need is to find the k stores with the minimal values as a function of the price and the distance from the person to the store. Motivated by this kind of applications with queries on both

location-dependent and location-independent attributes, we propose the heterogeneous k-nearest neighbor (HkNN) query.

Example 2. Consider another example for H*k*NN queries over multi-dimensional data. In a digital library, each subscriber has a profile that records the degrees of his/her preference in various fields. Each profile can be considered as an object in a multidimensional space in which every dimension stands for a field. Also, each profile has several location-independent attributes, like income, age, and so on. If a service provider would like to publish a new electronic magazine focusing on young people, an H*k*NN query that retrieves the best *k* subscribers with smaller ages and profiles similar to the magazine content will be launched. The H*k*NN query here is to find the *k* subscribers with the minimal values as a function of the age and the distance from the profile to the magazine content.

Previous work on monitoring continuous *k*-nearest neighbor queries over moving objects can be broadly divided into two categories. In the first category, the motion patterns of objects are assumed to be known or predictable [2][5][7]. The second category does not assume the motion patterns of objects [3][4][8][9]. There are two reasons that the previous approaches to the *k*NN problem are not suitable for the H*k*NN query: (1) an intuitive way is to regard the location-independent attribute as an extra dimension in the multidimensional space constituted by the location-dependent attributes. After that, the total cost formula is used as the distance function in order to apply the previous approaches. Unfortunately, mixing two kinds of attributes with different domain sizes makes it difficult to build an index for efficient query processing. Moreover, if the operation "–", "*", or "/" is adopted in the total cost formula, the distance function will not satisfy the triangle inequality and the pruning techniques will fail. More detailed discussions in this aspect can be found in [6]. (2) The previous approaches cannot deal with different operations with a single index. Therefore, a generic framework supporting the four operations and an efficient index structure are required.

In this paper, we propose a generic framework for efficiently processing the H*k*NN queries over moving objects and focus on the H*k*NN problem in which only one location-independent attribute is considered. The formulas in different applications to compute the total cost for each object can be generalized as *total cost* = *V_COST* (from the location-independent attribute) ***op*** *D_COST* (from the location-dependent attributes), where the operator ***op*** can be one of the four operations "+", "–", "*", or "/", as user needs. Moreover, we consider the situations like Example 2, in which both the location-dependent and location-independent attributes of objects may change with time. In the remainder of this paper, for ease of presentation, we illustrate our methods only for the H*k*NN queries with ***op*** "+". The considerations for the other ***op***s ("-", "*", "/") are similar and can be found in [6].

Our method of H*k*NN query evaluation proceeds in two steps. In the first step, arbitrary *k* objects near the query are selected into the answer set. For each of the *k* objects, the total cost is computed and the one with the worst total cost is called the *target object*. Since these *k* objects may not be the correct answers, we then check if any other object has a total cost better than that of the target object. Based on a *bounding property*, the total cost of the target object is used to compute two bounds - *value bound* and *distance bound*, which respectively indicate the upper bounds of the V_COST and D_COST for all the objects with better total costs. In other words, these bounds limit the search space of the H*k*NN query to a *safe region* for the remaining objects, in which all the objects with better total costs are located. In the second step,

only the objects inside the safe region are retrieved and checked. Once an object with a better total cost is found, it is added into the answer set with the target object discarded. After that, a new target object comes out and the two bounds can be lowered to further reduce the search space. The second step is repeatedly executed until there is no object in the safe region. In this way, a large amount of unnecessary computation can be avoided. The proposed framework also supports continuous H*k*NN queries for a dynamic environment where the objects may continuously update their location-dependent or location-independent attribute values. In our approach, while receiving an object update, only the queries that can be affected are reevaluated and thus unnecessary computation on irrelevant queries can be avoided.

The rest of the paper is organized as follows. In Section 2, basic definitions and data structures are described. Section 3 depicts the techniques for H*k*NN query processing. Section 4 illustrates how to handle the updates with our index structure in a dynamic environment. Section 5 shows the experimental results and Section 6 concludes the paper with some future works depicted.

2. Basic Definitions and Data Structures

2.1 Basic Definitions

In this paper, we address the H*k*NN problem involving only one location-independent attribute. In the following, we first define several terms and the H*k*NN problem. Table 1 shows the symbols and functions used in this section.

Table 1. Notation and their definitions

Notation	Definitions
O	The set of moving objects
o	A moving object in *O*
q	The heterogeneous *k*-nearest neighbor query
dist(o, q)	The Euclidean distance between object *o* and query *q*
op	The user-defined operator, can be +, -, *, *or* /

Definition 1: (Object and query representations) A moving object *o* is represented as $o(v, p)$, where *v* denotes the value of its location-independent attribute and $p=<c_1, c_2,\ldots c_n>$ denotes its coordinates, i.e., the location-dependent attribute of *o* (*n* is the number of dimensions). An H*k*NN query *q* is represented as $q(op, p')$, where *op* denotes one of four operators and $p'=<c'_1, c'_2, \ldots c'_n>$ denotes its coordinates, i.e., the location-dependent attribute of *q*.

Definition 2: (D_COST) Let *d-factor* be the cost for one distance unit. The cost of the location-independent attribute for an object $o(v, p)$ with respect to query $q(op, p')$ is defined as: *D_COST = dist(o, q) * d-factor*.

Definition 3: (V_COST) The cost of the location-independent attribute for object $o(v, p)$ is defined as: *V_COST = v*.

Definition 4: (T_COST) Given object *o(v, p)* and H*k*NN query *q(op, p')*, we refer to the total cost of *o* with respect to *q* as: *T_COST(o, q) = V_COST op D_ COST*.

Most applications in the real world demand the objects that are closest to a given query. Therefore, our framework is designed to prefer the objects with smaller values of D_COST no matter which operator is adopted. Applications with **op** "+" and "*" also have the nature to prefer objects with smaller values of V_COST and their goal is to find the k objects with the smallest values of T_COST. In contrast, applications with **op** "-" and "/" require objects with larger values of V_COST but smaller values of D_COST and their goal is to obtain the k objects with the largest values of T_COST. As a result, the definition of the HkNN problem is given as below by considering the two classes of operators.

Definition 5: (HkNN) Given a set of moving objects O, (1) if op of query q is "+" or "*", the **HkNN** answers of q are defined as: $HkNN(q)=\{o \in O | T\_COST(o, q) \leq T\_COST(o_k, q)\}$, where o_k is the object with the k-th smallest value of T_COST in O. (2) If op of q is "-" or "/", the HkNN answers of q are: $HkNN(q)=\{o \in O | T\_COST(o, q) \geq T\_COST(o_k, q)\}$, where o_k is the object with the k-th largest value of T_COST in O.

In this paper, the d-factor is set as 1 for simplicity and, from now on, we illustrate our approach using the case that the operator of the launched query is "+". The proposed framework is developed for a general environment, i.e., the motion patterns of objects and queries are unpredictable. The following illustration uses 2D data, but the proposed approach can be applied to the environment with arbitrary dimensionality.

2.2 Data Structures

Before introducing our approach, we first present four data structures that will be used.

Object Table. Each object o is associated with a set of attributes, including the object ID id_o, the location-dependent attribute p_o, the location-independent attribute v_o, and a set S_o of queries the answer sets of which contain o.

Hierarchical Aggregate Grid Index. Previous work [9] uses the hierarchical grid structure to reduce the performance degradation caused by skewed data. Differently, we here adopt it to speed up the HkNN query processing by enclosing location-independent attribute information in each cell of the hierarchical grid structure. Note that we can also embed this kind of information in each internal node of an R-tree structure in a similar way. In this paper, we adopt the grid-based structure because of its efficient construction and maintenance in a dynamic environment.

The hierarchical aggregate grid index has several levels of grids and consists of two types of cells: *basis cells* and *index cells*. Index cells form a hierarchy, where each index cell points to smaller cells it covers at the lower level (called the *sub-cells*). The bottom level of the hierarchical grid structure is composed of basis cells, the smallest unit in the index. Let $C_{i,j}$ denote a cell, which can be an index cell or a basis cell, at column i and row j of grid level C. Moreover, each basis cell $X_{i,j}$ (Assume the bottom level is level X) with equal length δ is associated with one bucket that stores every object ID with coordinate (x, y), where x is in the range $[i\delta, (i+1)\delta]$ and y is in the range $[j\delta, (j+1)\delta]$. Every object or query moving from (x_{old}, y_{old}) to (x_{new}, y_{new}) is deleted from the bucket of cell $(\lfloor x_{old}/\delta \rfloor, \lfloor y_{old}/\delta \rfloor)$ and inserted into the bucket of cell $(\lfloor x_{new}/\delta \rfloor, \lfloor y_{new}/\delta \rfloor)$. In addition, to enclose location-independent attributes of objects in each cell, both the basis cell and the index cell are associated with three pieces of aggregate information: *min*, *max*, *and count*. For a basis cell, the *min* (*max*) indicates

the minimal (maximal) value of location-independent attributes for the objects in the cell and similarly the *count* represents the total number of objects enclosed in the cell. For an index cell, *min* (*max*) is the minimum (maximum) of all the *min* values attached on its sub-cells, while the *count* keeps the sum of all the *count* values attached on the sub-cells. Figure 1 shows an object table and a 3-D illustration of the hierarchical aggregate grid index. Both types of cells are associated with the aggregate information (*a*, *b*, *c*), where *a*, *b*, and *c* represent *min*, *max* and *count*, respectively. Moreover, every basis cell is associated with an object bucket storing the IDs of the enclosed objects together with the links, pointing to the corresponding entries in the object table.

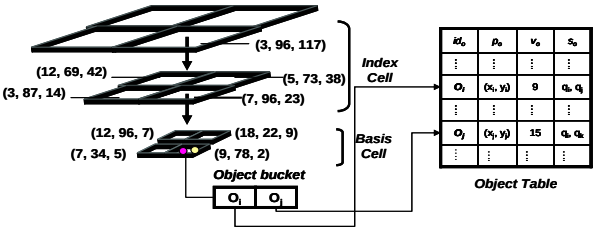

Fig. 1. Hierarchical aggregate grid index and object table

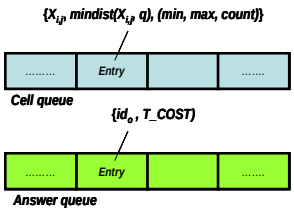

Fig. 2. Cell queue and answer queue

Cell Queue. Let *mindist*($X_{i,j}$, *q*) be the minimal distance between cell $X_{i,j}$ and query *q*, i.e., the minimal possible distance between any objects in cell $X_{i,j}$ and *q*. Whenever a query *q* is evaluated (or reevaluated), a cell queue *CQ* is created. Each entry in *CQ* stores a cell $X_{i,j}$ (index cell or basis cell), *mindist*($X_{i,j}$, *q*), and the aggregate information: min, max, count of $X_{i,j}$. The entries in *CQ* are kept in ascending order of *mindist*($X_{i,j}$, *q*).

Answer Queue. Each query *q* is associated with an answer queue *AQ* to maintain the current set of query answers. Each entry in the answer queue keeps an object ID and its T_COST with respect to *q*. The entries in *AQ* are kept in ascending order of T_COST. Figure 2 shows both kinds of queues for query *q*.

3. Efficient Evaluation of H*k*NN Queries

3.1 Problem Characteristic and Approach Overview

A naive method for processing the H*k*NN query is to compute the total cost of each object and then select the *k* objects with the smallest values as the answers. Nevertheless, it is inefficient if the number of objects is very large. Consider an alternative method using the data structures in Section 2.2. To evaluate query *q*, all the cells in the hierarchical aggregate grid structure can be pushed into the cell queue *CQ* in ascending order of their minimal distances from *q*. From *CQ*, the cells are explored one by one to obtain the first *k* objects. The *k* objects are then put into the answer queue *AQ* in ascending order of their total costs. We call the object with the largest total cost as the *target object* and its total cost as the *target cost*. Given *AQ* and the target cost, we apply the following property to prune the remaining objects in *CQ*, which cannot be the H*k*NN answers.

Property 1: (Bounding property) *For an object o in cell $X_{i,j}$ of CQ, if its total cost is not larger than the target cost, then its D_COST is not larger than the target cost and its V_COST is not larger than the target cost minus mindist($X_{i,j}$, q).*
Proof: *Let r be the target object. Moreover, let D_z, V_z and T_z respectively denote the D_COST, V_COST and T_COST of object z. Since $V_o \geq 0$ and $T_o=D_o+V_o \leq T_r=D_r+V_r$, we have that $D_o \leq D_r+V_r-V_o \leq D_r+V_r-0=T_r$. Similarly, because $D_o \geq mindist(X_{i,j}, q)$, we have that $V_o \leq D_r+V_r-D_o \leq D_r+V_r-mindist(X_{i,j}, q)=T_r-mindist(X_{i,j}, q)$.*

From this property, two upper bounds, *value bound* and *distance bound*, are obtained and used to prune the cells in *CQ*, in which all the objects violate either of them. The objects in a cell are retrieved to compute their total costs if and only if the cell satisfies both bounds. *AQ* and the target object are then updated. Since the new target object results in a smaller target cost, the bounds can be tighter and tighter. Eventually, *CQ* will become empty and *AQ* will have the *k* objects with the smallest total costs. The pruning mechanism using the two bounds will be described in Section 3.3.

3.2 Step 1: Retrieving the First *k* Objects

Given an H*k*NN query *q*, we discuss our first step of query evaluation for *q* in this section. Let *CQ* and *AQ* respectively denote the cell queue and the answer queue of *q*. To begin with our approach, the cells at the highest level of the index are first visited and then inserted into *CQ* in ascending order of their minimal distances from *q*. Next, our approach starts to retrieve the first entry in *CQ*. If it is an index cell, its sub-cells are inserted into *CQ* according to their minimal distances to *q*. This process is repeated until the first entry of *CQ* is a basis cell. In this case, the total costs of all the objects in this cell are computed. Their object IDs and total costs are then inserted into *AQ* in ascending order of their total costs. This step terminates while the number of objects in *AQ* is not less than *k*. Note that if the number of objects in *AQ* is more than *k*, only the *k* objects with the smallest total costs are chosen and used in the next step.

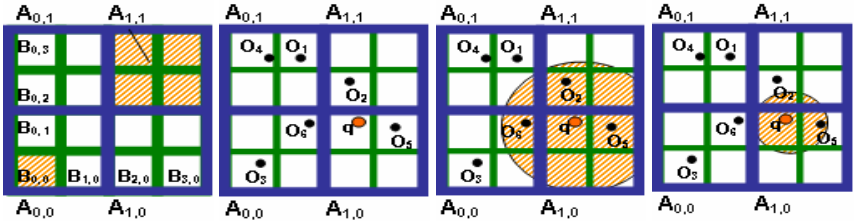

Fig.3. An example of query evaluation using two-level hierarchical aggregate grid index

Example 3. Figure 3 shows an example of query evaluation using a two-level hierarchical aggregate grid index. Let *A* and *B* represent the higher level and the lower level of grids, respectively. The bigger shaded cell ($A_{1,1}$) and the smaller one ($B_{0,0}$) are shown in Figure 3 (a). In Figure 3 (b), given an H2NN query *q*, the search for the first *k* objects is executed (let *k* be *2*). Initially, the cells of the higher level are added into *CQ* in ascending order of their minimal distances from *q*, i.e., $Q=\{A_{1,0}, A_{1,1}, A_{0,0}, A_{0,1}\}$. Then, the first entry $A_{1,0}$ is retrieved and its sub-cells $B_{2,0}$, $B_{2,1}$, $B_{3,0}$, $B_{3,1}$ are inserted into *CQ*. Since all the counts in $B_{2,0}$, $B_{2,1}$, and $B_{3,0}$ are zero, these cells are ignored to avoid unnecessary computation. After that, *CQ* becomes $\{A_{1,1}, B_{3,1}, A_{0,0},$

$A_{0,1}$}. Next, $A_{1,1}$ is retrieved in the same way to update CQ as {$B_{2,2}$, $B_{3,1}$, $A_{0,0}$, $A_{0,1}$}. $B_{2,2}$ is then retrieved and the total cost of o_2 is computed (let its total cost be *1.6*). As a result, o_2 is inserted into AQ with its total cost, i.e., AQ ={*(o_2,1.6)*}. This step terminates after the next entry $B_{3,1}$ updates AQ as {*(o_5,0.7)*, *(o_2,1.6)*}. At this time, CQ becomes { $A_{0,0}$, $A_{0,1}$}.

3.3 Step2: H*k*NN Search with Pruning Mechanism

The second step for query evaluation is to iteratively find the objects with smaller total costs and replace the target object to further reduce the search space until the safe region is empty. A naïve method is to sequentially examine all the objects located in the circle centered at q with radius D (for short, from now on, sometimes we denote the distance bound and value bound as D and V, respectively). Obviously, this method can be inefficient if more basis cells than necessary are checked. Therefore, we design a pruning mechanism to skip the objects that are unable to be the final answers. The pruning mechanism keeps retrieving the first entry of CQ to check whether any object in it satisfies the two bounds. The following are the two pruning techniques we adopt.

Pruning by Distance Bound. Cell $X_{i,j}$ in CQ needs to be checked if and only if it overlaps with the circle centered by q with radius D, i.e., *mindist* ($X_{i,j}$, q) ≤ D. In this case, cell $X_{i,j}$ may contain an object whose distance from q is less than D and therefore should be further checked by the value bound. Otherwise, $X_{i,j}$ and all the remaining cells in CQ can be discarded because all the objects inside them cannot be the final answers (their distances from q must be larger than D).

Pruning by Value Bound. For each cell $X_{i,j}$ in CQ, the value bound is computed as V =c_t – *mindist*($X_{i,j}$, q), where c_t denotes the target cost. If the *min* value associated with $X_{i,j}$ is larger than V, $X_{i,j}$ can be omitted. Otherwise, there are two cases to consider. If $X_{i,j}$ is an index cell, its sub-cells are retrieved from the hierarchical aggregate grid structure and inserted into CQ. On the other hand, if $X_{i,j}$ is a basis cell, the total costs of all objects in it are computed and then the objects with total costs smaller than c_t are inserted into AQ. Finally, only the k objects with the smallest total costs are left in AQ and then a new target object can be found. In this way, new and smaller distance bound and value bound will be obtained.

Example 4. Figure 3(c) and 3(d) show an example for the second step of query evaluation. We assume that *mindist($A_{0,0}$, q)*, *mindist($B_{1,1}$, q)*, and dist(o_6, q) are 0.8, 0.8, and 0.9, respectively. In addition, we assume that the min value of $A_{0,0}$ and $B_{1,1}$ are 0.6 and 0.4, respectively and the V_COST of o_6 is 0.1. In Figure 3(c), the radius of the shaded circle centered by q is equal to D. Following Example 3, AQ is {*(o_5,0.7)*, *(o_2,1.6)*} and. $A_{0,0}$ is first examined. The value bound for $A_{0,0}$, i.e., *1.6 – mindist($A_{0,0}$, q)*, is larger than the *min* value of $A_{0,0}$, its sub-cells are inserted to CQ. Again, since $B_{0,1}$ and $B_{1,0}$ are empty, CQ becomes {$B_{1,1}$, $A_{0,1}$, $B_{0,0}$}.The value bound of $B_{1,1}$ is computed in the same way and equals to *0.8*, which is larger than the *min* value of $B_{1,1}$. Therefore, the object o_6 in $B_{1,1}$ is retrieved to replace o_2 as the new target object (o_6 has a total cost smaller than that of o_2), i.e., AQ ={*(o_5,0.7)*, *(o_6,1)*}. Then, the shaded circle shrinks because D decreases. We show the new circle in Figure 3(d). Next, we get the next entry $A_{0,1}$. From Figure 3 (d), we can observe that it does not overlap with the shaded circle. In other words, the minimal possible distance of any objects in $A_{0,1}$

is larger than D and therefore the remaining cells in CQ can be skipped. The pruning process ends and the final answers of q are $\{o_5, o_6\}$.

4. Continuous Update of H*k*NN Query Answers

Another contribution of our framework is the update mechanism for maintaining the answers of queries in a dynamic environment, where objects and queries may update their status continuously. For objects, both of their positions and location-independent attribute values can be updated, while for queries, only their positions may be updated. In the following, we illustrate our approach for continuously processing a single object update.

To maintain the H*k*NN query answers while receiving an object update, a naïve way is to reevaluate all the H*k*NN queries. However, it is impractical if there are a large number of queries registered. Obviously, a better solution exists if we can find out all the queries that will be influenced by the updated object. Let $cost_k$ denote the largest total cost of the object in the answer queue of q after the query evaluation. Queries whose answers are affected by the updated object can be divided into two classes. In the following, we discuss them respectively.

Case 1. The first class consists of the queries whose answer sets do not include the updated object. For a query q belonging to this class, its $cost_k$ will not be changed immediately in response to the updated object. Based on the bounding property, an object o_i can be used to replace the target object of q only if both the D_COST and V_COST of o_i satisfy the distance bound and value bound of q, respectively. Motivated by this, for each query, we refer to the *distance influence region* as a region where all the objects satisfy the distance bound of q. Similarly, we define the *value influence region* to identify all the objects satisfying the value bound of q. The total cost of o_i is compared with $cost_k$ only if the two regions of query q both contain o_i. If object o_i has a smaller total cost, it is a new answer of q and $cost_k$ is updated. Otherwise, the answer queue of q remains unchanged. More details of the two regions are discussed in the following:

Distance Influence Region. The region composed of all the basis cells that intersect the circle centered at q with radius D is called the *distance influence region*. Clearly, a basis cell may be covered by multiple distance influence regions of different queries. To identify these queries, each basis cell of the grid is associated with a query bucket containing the queries whose distance influence regions intersect it. Figure 4 shows this data structure.

Value Influence Region. We adopt the B+-tree, named *the value influence region tree*, to organize the value influence regions of all the queries. Figure 5 shows an example. In this tree, the value bounds of queries are the access keys attached on the nodes. Moreover, each internal node (or called *index node*) is constructed after the split or merging of tree nodes. On the other hand, each leaf node (or called *data node*) keeps a set of queries. Note that each leaf node has a pointer, pointing to the sibling next to it. Assume that there is an object o_i with V_COST = 5. To find the queries whose value influence regions contain object o_i, a range search for all the queries with $V \geq 5$ is launched on the value influence region tree. Since the amount of returned queries may be huge, we first check the query bucket of the cell containing the updated objected to find the queries whose distance influence regions contain o_i. Then,

these queries are checked one by one to see whether their value influence regions contain o_i by launching exact searches on the value influence region tree.

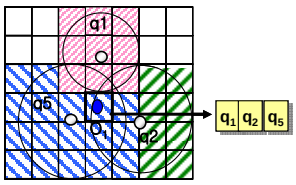

Fig. 4. Distance influence region

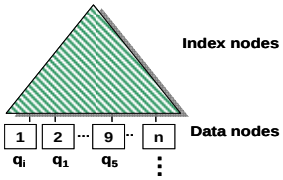

Fig. 5. Value influence region tree

Case 2. The second class includes the queries that answer queues of which contain the updated object. In this case, $cost_k$ may be changed even when the answer queue has the same set of objects. According to the possible changes of $cost_k$, there are three situations to consider: The first situation is that $cost_k$ remains unchanged. This may occur due to two causes: (1) the updated object is not the target object and its new total cost is still not larger than $cost_k$ or (2) the updated object is the target object but its new total cost is unchanged although both the V_COST and D_COST are changed. For both causes, we only need to update the information about the updated object. The second situation is that $cost_k$ becomes smaller. This also occurs due to two causes: (1) the updated object is the target object and its total cost decreases. or (2) the updated object is the target object and its new total cost is smaller than the object with *k-1*th smallest total cost in the answer set of q. Since both $cost_k$ and the order of answers may be changed, we need to update the two bounds and the two influence regions by using the new target object and $cost_k$.

The third situation is that $cost_k$ becomes larger. It happens in two conditions: (1) the updated object is the target object and the changed V_COST or D_COST makes its total cost larger or (2) the updated object is not the target object but belongs to the answer set of q, and its updated V_COST or D_COST let its total cost larger than $cost_k$. In both conditions, since $cost_k$ becomes larger, the object with the *k+1*th smallest cost may replace the updated object as a new answer of q. Therefore, we assume the updated object, denoted as o_t, to be the target object and invoke the H*k*NN reevaluation to find the object with smaller total cost than o_t to replace it. Due to space limitation, more details about the H*k*NN reevaluation are discussed in [6].

Example 5. We illustrate an example for processing an object update using Figure 4 and Figure 5. Suppose the server receives an update record $\{o_1, x, y, v, x', y', v'\}$ representing o_1 moves from (x, y) to (x', y') and its value changes from v to v'. Assume $v' = 3$ and the updated $dist(o_1, q) = 15$. First, o_1 is deleted from the object bucket of the basis cell $X_{i,j}$ and then inserted into the basis cell $X_{i',j'}$, where i, j, i', and j' are $\lfloor x/\delta \rfloor, \lfloor y/\delta \rfloor, \lfloor x'/\delta \rfloor$, and $\lfloor y'/\delta \rfloor$, respectively. Next, the new total cost of o_1 is computed. After that, the queries whose answer sets contains o_1 are found by checking the object table. Assume q_2 is found and the total cost of the target object of q_2 is 13. Since the new total cost of o_1 is 18, the $cost_k$ of q_2 becomes larger (from 13 to 18, the third situation of Case 2). Therefore, H*k*NN query reevaluation is invoked to update the answers of q_2. Next, the algorithm finds the queries whose distance and value influence region both contain o_1 but their answer sets do not include o_1 (Case 1). As shown in Figure 4, the cell enclosing o_1 is covered in the distance influence regions of q_1, q_2, and q_5. Since query q_2 has been processed, it can be ignored here. Next, we checks if the value influence region of q_2 or q_5 contains o_1. A range query q_r, which

searches the queries with $V \geq 3$ on the value influence region tree is issued. In Figure 5, because q_1 is linked to the leaf node with key = 2, i.e. the value bound of q_1 is 2, q_1 will not be in the returned results of q_r. Thus, o_1 is not in the value influence region of q_1. On the contrary, q_5 with $V = 9$ is in the results of q_r. Since o_1 is contained in both the distance influence region and value influence region of q_5, o_1 is probably qualified to be a new answer of q_5. Therefore, the updated total cost of o_1 ($cost_1$, for short) needs to be compared with that of the target object of q_5 ($cost_e$, for short). If $cost_1 < cost_e$, the target object of q_5 is deleted from the answer set of q_5 and o_1 becomes a new answer. Then, the bounds and influence regions for q_5 are updated. Otherwise, q_5 is not affected by the update of o_1.

5. Experiments

We compare the proposed method using the hierarchical aggregate grid index with a method using a one-level grid and a brute force method. For simplicity, we call the above three methods *HAG*, *OLG*, and *BF*, respectively. Similar to HAG, OLG is a method based on the bounding property proposed in this paper. The distinction of OLG from HAG is that it employs one-level grid structure instead of hierarchical grid index. One-level grid structure consists of equal-sized cells and each cell is associated with three pieces of aggregate information-min, max and count. In the first phase of evaluating an H*k*NN query *q*, OLG adopts the method proposed in [9] to find the *k*NN of *q* as the first *k* objects. Next, it pushes the cells overlapping with the circle centered by *q* with radius *D* into the queue of *q* and runs the pruning algorithm in the same way as our approach. The brute-force method for comparison computes the total costs for all objects in the database and then choose the *k* objects with smallest total costs as the answers.

In all the experiments, we utilize a program modified from the *Network-based Generator of Moving Objects* [1] to generate a set of moving objects and queries. The generator outputs a set of objects with their values at every timestamp. The default mobility of objects and queries are 10% and 5%, respectively. Table 2 lists the parameters of the data sets, where the default values are bold and italic. All our experiments are performed for 100 time-stamps and the CPU time (in seconds) is reported after a work is completed. For HAG and OLG, the locations of objects (or queries) and the aggregate information of all cells are updated after receiving all the update records at each timestamp.

Table 2. Parameters of the data sets

Parameter	Range
Number of Object (K)	10, 50, ***100***, 150, 200
Number of Queries (K)	1, 2, 4, ***6***, 8, 10

Table 3. Grid structures with various cell sizes

OLG	HAG
32×32	4×4 ,16×16,32×32
64×64	4×4, 16×16, 64×64
128×128	4×4, 32×32, 128×128
256×256	4×4, 32×32, 256×256
512×512	4×4, 64×64, 512×512
1024×1024	4×4, 64×64, 1024×1024

5.1 Experimental Results

The first experiment evaluates how the performance of OLG and HAG will be affected by different grid size. Moreover, for HAG, the number of levels in the hierarchical grid index also has a great impact on its performance. To fairly compare the performance of the two methods with respect to the grid size, we fix the number of

levels to 3 for HAG and vary the grid size of both methods from 32×32 to 1024×1024. For OLG, this parameter indicates the grid size of the one-level grid index, while for HAG it represents the grid size for the grid at basis level in the hirarchical aggregate grid index (denoted as $2^b\times2^b$). In addition, we fix the grid size of the top level to 4×4 (i.e. $2^2\times2^2$). The granularity of any index cell at a lower level is set to be $2^l\times2^l$, where $l=\lceil(\log_2 b+2)/2\rceil$. Table 3 summarizes the granularities used in our experiments.

The experimental results are shown in Figure 6. Both methods achieve the best performance when the 128×128 grid is used, while the other grid sizes cause the two methods higher CPU costs. This is because their grid indices with fine granularities incur frequent updates, whereas coarse granularities result in linear searches on huge object buckets during accessing a cell. We also observe that 3-level HAG has better performance than OLG in all cases. The reasons are as follows: (1) After the first step of evaluation of query q, OLG puts all the cells intersecting with the circle centered at q with radius D into the cell queue in ascending order of their minimal distances from q. The computations of these minimal distances for all cells covered by the circle result in expensive costs. (2) Despite HAG have the same number of cells at basis level as that of the index in OLG, the good pruning effect of HAG helps pruning many index cells that do not cover any answer. Thus, unnecessary computations for minimal distances for all basis cells can be avoided. (3) Moreover, although HAG requires more updates for maintaining aggregate information due to multiple-level grids, such updates (in both methods) can be done by batch processing at each time-stamp. Therefore, the cost of updating hierarchical aggregate information can be limited. Based on the above observations, the remaining experiments are made using the 128×128 grid for HAG and OLG.

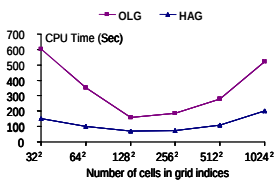

Fig. 6. Performance vs. Granularity

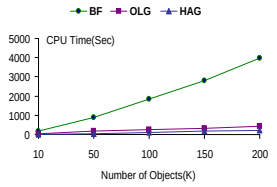

Fig. 7. Performance vs. N_O

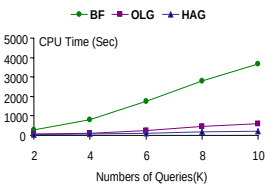

Fig. 8. Performance vs. N_Q

In the second experiment, we compare the performance of HAG, OLG and BF by varying the number of objects (denoted as N_O) from 10K to 200K. Figure 7 depicts that the CPU costs of all methods increase when N_O increases. Moreover, BF is much less efficient than the other two methods. The reason is that for each query, BF computes the total costs, in which expensive distance computations are involved, for all objects in the database. Furthermore, all the queries registered in the system are re-evaluated to keep the answers correct when the update requests are received. On the contrary, in all cases the execution times of OLG and HAG are relatively small. This is because both methods are designed based on the bounding property proposed in this paper, which helps greatly reducing the search space. Furthermore, they handle updates only on the queries whose answers may be affected by the updated objects instead of reevaluating all the registered queries. In Figure 7, we also observe that the performance of HAG is only a little better than OLG. The reason is that a larger N_O leads to the larger probability of a query reevaluation after the given object update. (The third situation in Case 2 for handling object updates) Similar result is shown in Figure 8, where the number of queries (denoted as N_Q) is varied. HAG has better

performance than OLG. As mentioned in the first experiment, HAG possesses several advantages over OLG in processing a single query. Therefore, OLG is also more sensitive than HAG when N_Q increases. To sum up, in both figures, HAG is more efficient than OLG and BF. These experimental results verify that HAG achieves good scalability and efficiency.

6. Conclusion and Future Works

In this paper, we introduce the *heterogeneous k-nearest neighbor (HkNN)* query, a new paradigm considering both location-dependent and location-independent attributes over moving objects. H*k*NN queries are of nature interesting in many applications. To the best of our knowledge, this is the first work addressing the H*k*NN problem and providing an efficient approach for HkNN query evaluation. Based on the bounding property, our approach employs the *hierarchical aggregate grid index*, which recursively aggregates the values of location-independent attribute in a hierarchy of cells, to quickly reduce the search space of the H*k*NN query. Furthermore, we developed an efficient update mechanism for continuously monitoring the affected H*k*NN queries during an object update and for maintaining the correctness of query answers. In particular, our approach can handle different types of operators with a single index. Our experimental results demonstrate the efficiency and scalability of the proposed techniques.

In our work, we process every query using a cell queue that employs the minimal distance between the query and each enclosed cell as its key. In other words, we adopt a distance-based technique to solve the H*k*NN problem in this paper. In the future, we plan to research from the aspect of location-independent attributes and then develop a hybrid mechanism that can adaptively determine from which aspect the query processing should start with.

References

[1] T. Brinkhoff. A Framework for Generating Network-Based Moving Objects. *GeoInformatica*, 6(2): 153-180, 2002.

[2] G. S. Iwerks, H. Samet, and K. Smith. Continuous k-nearest neighbor queries for continuously moving points with updates. *VLDB*, 2003.

[3] M. F. Mokbel, X. Xiong, and W. G. Aref. SINA: Scalable incremental processing of continuous queries in spatio-temporal databases. *SIGMOD*, 2004.

[4] K. Mouratidis, D. Papadias, and M. Hadjieleftheriou, "Conceptual Partitioning: An Efficient Method for Continuous Nearest Neighbor Monitoring," *SIGMOD*, 2005.

[5] K. Raptopoulou, A. Papadopoulos, and Y. Manolopoulos. Fast nearest-neighbor query processing in moving –object database. *GeoInformatica*, 7(2):113-137, 2003.

[6] Y. C. Su. Technique Report: Monitoring Heterogeneous Nearest Neighbors for Moving Objects Considering Location-Independent Attributes. http://make.cs.nthu.edu.tw/people/Steffi/Technique.htm, 2006.

[7] Y. Tao, D. Papadias. Time-parameterized queries in spatio-temporal databases. *SIGMOD* Conference, 2002.

[8] X. Xiong, M. F. Mokbel, and W. G. Aref. SEA-CNN: Scalable processing of continuous k-nearest neighbor queries in spatio-temporal databases. *ICDE*, 2005.

[9] X. Yu, K. Q. Pu, and N. Koudas. Monitoring k-nearest neighbor queries over moving objects. *ICDE*, 2005.