SIMkNN: A Scalable Method for In-Memory *k*NN Search over Moving Objects in Road Networks

Bin Cao, *Member, IEEE,* Chenyu Hou, Suifei Li, Jing Fan,Jianwei Yin Baihua Zheng, and Jie Bao

Abstract—Nowadays, many location-based applications require the ability of querying k-nearest neighbors over a very large scale of moving objects in road networks, e.g., taxi-calling and ride-sharing services. Traditional grid index with equal-sized cells can not adapt to the skewed distribution of moving objects in real scenarios. Thus, to obtain the fast querying response time, the grid needs to be split into more smaller cells which introduces the side-effect of higher memory cost, i.e., maintaining such a large volume of cells requires a much larger memory space at the server side. In this paper, we present SIMkNN, a scalable and in-memory *k*NN query processing technique. SIMkNN is dual-index driven, where we adopt a R-tree to store the topology of the road network and a *hierarchical grid model* to manage the moving objects in non-uniform distribution. To answer a *k*NN query in real time, SIMkNN adopts the strategy that incrementally enlarges the search area for network distance based nearest neighbor evaluation. It is far from trivial to perform the space expansion within the hierarchical grid index. For a given cell, we firstly define its neighbors in different directions, then propose a cell communication technique which allows each cell in the hierarchical grid index to be aware of its neighbors at any time. Accordingly, an efficient space expansion algorithm to generate the estimation area is proposed. The experimental evaluation shows that SIMkNN outperforms the baseline algorithm in terms of time and memory efficiency.

Index Terms—k-nearest neighbors, road network, hierarchical grid index.

1 Introduction

In this paper, we study the problem of finding the *snapshot* k nearest neighbors (kNN) of moving objects for a query point q in a road network. Nowadays, efficient snapshot kNN query processing over moving objects in road networks has become an essential building block for many location-based applications, e.g., both the ride hailing and the ride sharing services [1], [2] need to find the k nearest drivers, in terms of the network distance, for the passengers in real-time. Didi, the largest taxi-hailing company in China, claims that the total number of daily requests is approaching 10 million during peak hours [3]. This extreme large number of requests poses a big challenge to the spatial kNN query processing. The faster the result is located, the shorter the response time is, the higher the possibility that the returned result is still valid, and the sooner the incoming requests can be served. In other words, the capability of processing a large number of kNN queries simultaneously is critical to many real-time location-based applications. Consequently, we devote this paper to the problem of processing snapshot kNN searches over moving objects in a road network, and we assume in-memory database is available to maintain the index and the underlying road network because of the high performance and popularity of in-memory databases [4], [5].

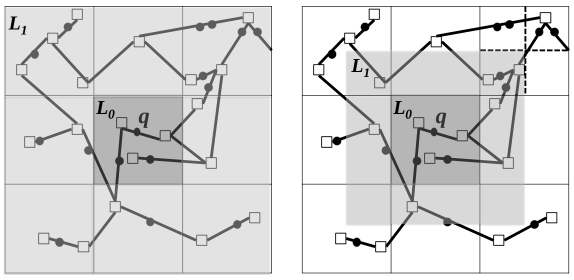
- B.Cao, C.Hou and J.Fan are with the College of Computer Science, Zhejiang University of Technology, Hangzhou, China
  E-mail: bincao@zjut.edu.cn, houcy@zjut.edu.cn, fanjing@zjut.edu.cn
- S.Li is with the Netease, Inc., Hangzhou, China. E-mail: hzlisuifei@corp.netease.com
- J.Yin is with the College of Computer Science, Zhejiang University, Hangzhou, China. E-mail: zjuyjw@cs.zju.edu.cn
- B.Zheng is with the School of Information Systems, Singapore Management University. E-mail:bhzheng@smu.edu.sg
- J.Bao is with the Microsoft Research Asia, Beijing, China. E-mail: jiebao@microsoft.com

Manuscript received April 19, 2016

Although the snapshot kNN search is static as it is based on a static query point, the underlying objects are moving. To be more specific, given a query q that is processed at time stamp t_i, we can only guarantee that the located k objects are nearest to q at time stamp t_i. However, if the processing of the query takes Δt, the result is only returned to the query issuer at time stamp $(t_i+\Delta t)$. The larger the Δt is, the higher the likelihood that the returned k objects are no longer the real k nearest ones as objects change their positions from t_i to $(t_i + \Delta t)$. Here, we name Δt as *staleness*, and ideally the value of staleness should be as small as possible. In order to achieve a small staleness value, the index structure that indexes the moving objects should support efficient update operations as underlying objects are expected to change their locations continuously; and the query search algorithm should support efficient kNN searches.

There are a few of existing approaches that try to reduce the value of staleness by taking the above two mentioned challenges into consideration, among which MOVNet [6], [7], [8] is a representative. It constructs a dual-index, including an on-disk R-tree [9] to store the road network and an in-memory grid index to deal with the frequent location updates from the moving objects. In addition, MOVNet restricts the network distance based processing of kNN to a small range instead of the whole network in order to address the efficiency issue. It loads the road network in a small search range via a range query over R-tree, and approximates the grid cells that contain at least k objects within this small search range based on Euclidean distances. The search space can be gradually expanded to finish the search.

MOVNet performs an equal partition to build the grid index. The simplicity of equal partition does facilitate the update operations. However, it also brings in challenges for



(a) Grid based expansion (b) $\mathcal{H}$-grid based expansion

Fig. 1. Space expansion for estimating result space

query processing. It is well-known that according to Pareto principle, 80% of the traffic happens within 20% of the road network. In other words, the distribution of moving objects is expected to be skew [10], [11], [12]. Consequently, the real capacity of different grid cells is expected to be very different which significantly deteriorates the query processing performance [13]. An example is plotted in Fig. 1(a). Assume a 4NN query is issued at the query point q. Initially, the search space L_0 (i.e., the dark grey area) only covers the current grid cell where q falls within. However, as there are only three objects in L_0, the search space has to be expanded to L_1 depicted as the light grey area in Fig. 1(a) and in total 16 objects are retrieved. In order to locate the 4 nearest objects, we need to find the network distances from q to all the 16 objects which is very expensive. In order to reduce the number of network distance computations, we would like to limit the area expanded. If we are able to expand L_0 to L_1 depicted in Fig. 1(b), the total number of distance calculations could be reduced from 16 to 6. However, the equal-sized grid cells only support grid-cell-based expansion, i.e., the initial search range covering 1×1 grid cell will be expanded to 3×3 cells, and then to 5×5 cells, and so on. Consequently, the number of distance calculations and the size of search range could be reduced if the grid cell size becomes smaller. However, the smaller grid cell size increases the total number of grid cells which increases the storage overhead for grid index. Consequently, there is a tradeoff between the storage overhead and the query performance.

Alternatively, we propose SIMkNN, a **S**calable method for **I**n-**M**emory ***k*NN** search over moving objects in road networks as an enhanced version of MOVNet. The main improvement is introduced by the hierarchical grid index (in short $\mathcal{H}$-grid) which performs an uneven grid partition. $\mathcal{H}$-grid splits a cell into sub-cells when the number of objects exceeds its capacity, which shares the similar mechanism as *Quad-trees* and other hierarchical structures [14]. Hence, $\mathcal{H}$-grid [15] index partitions the grid cells based on the popularity of the sub-region covered by grid cells, and it partitions a cell into multiple smaller sub-cells if the number of objects inside the cell is larger than a threshold. Suppose the cell capacity is 4, the cell in the top-right corner of Fig.1(b) will be split into 2×2 sub-cells (denoted by the dotted lines). Grid cells under $\mathcal{H}$-grid are capacity-even but size-uneven. This means when we need to expand the search area during kNN searches, all the neighboring cells of the current search area share the same capacity but different sizes. For those denser regions, they will be covered by more grid cells, as compared with normal grid index. In other words, kNN search will evaluate the denser regions gradually. Take the top-right cell of Fig.1(b) as an example. The expansion will reach the light grey colored sub-cell first. Only when the current expanded search region still does not bound enough objects, the search range will be expanded further. Consequently, $\mathcal{H}$-grid is expected to reduce the number of distance calculations and improve the search performance.

Though the idea of $\mathcal{H}$-grid index is simple and not totally new, how to apply it to SIMkNN is not trivial and there are two main challenging issues we have to address.

- How to locate the neighbouring cells for a given cell in $\mathcal{H}$-grid index? A fundamental building block for the expansion of the search area is to locate the neighbouring cells of a given cell. Unlike traditional grid index, cells in $\mathcal{H}$-grid have different sizes and they are located in the different levels of the hierarchical index structure. We name this challenge as *cell locating*. To the best of our knowledge, there is no existing work that can support cell locating.
- How to expand the search space based on $\mathcal{H}$-grid index? If the space is partitioned into equal-sized grid cells, the expansion of the search space is guided by the side length of each grid. However, $\mathcal{H}$-grid has heterogeneous setting where cells have different sizes. We need to propose mechanisms to evaluate the advantages of different cell side length and to efficiently find the optimized side length for search space expansion.

To enable cell locating, we propose a *cell communication* technique to make sure each cell in $\mathcal{H}$-grid has the up-to-date knowledge of its neighbours. The communication is triggered whenever a cell merge or a cell split operation takes place. The new cells will find out their neighbours and meanwhile inform their neighbours of their arrival. Because of the complete and up-to-date knowledge of neighbouring cells, SIMkNN enables a more conservative search space expansion by expanding the search space to the smallest cell among the neighbours. This conservative expansion strategy helps to improve the search performance significantly. To sum up, we have made mainly four contributions in this paper.

1) We propose SIMkNN, a scalable in-memory processing method for snapshot kNN queries over moving objects in road networks.
2) We present a cell communication technique which makes sure that each cell of $\mathcal{H}$-grid index is aware of the up-to-date neighboring cells. To the best of our knowledge, we are the first to solve the cell locating problem for $\mathcal{H}$-grid index.
3) We introduce an efficient space expansion method to estimate the result space for kNN query based on the cell communication technique in $\mathcal{H}$-grid index.
4) We build a prototype for SIMkNN and demonstrate its super scalability and efficient search performance through an extensive experimental evaluation study.

The remainder of this paper is organized as follows. Section 2 presents preliminaries for SIMkNN. Section 3 introduces a cell communication technique based on the $\mathcal{H}$-grid

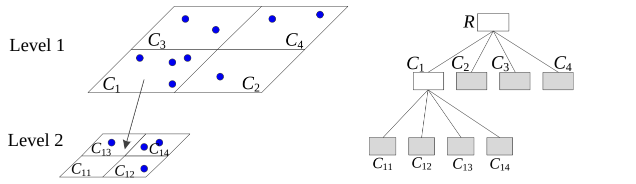


Fig. 2. A sample $\mathcal{H}$-grid index with the split factor $m = 2$ and the capacity $N_c = 4$

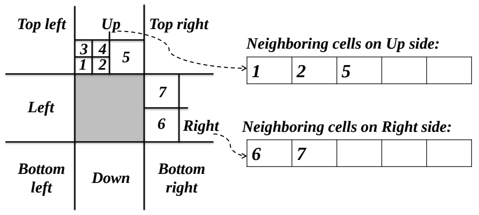


Fig. 3. Neighbors in eight directions for a given cell

index. Section 4 describes the details of query processing on top of $\mathcal{H}$-grid index, including the conservative search space expansion strategy. Section 5 reports the results of experimental evaluation and Section 6 reviews related work. Finally, Section 7 concludes this paper with some directions for future work.

2 Preliminaries

In this section, we first present a brief introduction to the $\mathcal{H}$-grid index and then introduce several terms related to cell neighbours that will be commonly used in the rest of this paper.

2.1 $\mathcal{H}$-grid Index

$\mathcal{H}$-grid index is an improved grid index where a cell capacity parameter N_c and a split factor m are introduced. Similar to other hierarchical indices such as Quad-tree [14], a cell that has too many objects and exceeds its capacity N_c will be split into $m \times m$ equal-sized smaller sub-cells. Initially, the entire search space is partitioned into $m \times m$ equal-sized grid cells, and the original $\mathcal{H}$-grid index only indexes those $m \times m$ grid cells. Thereafter, we insert the objects into corresponding grid cells based on their locations. A split operation is triggered when the number of objects associated with a cell C_i exceeds N_c. Accordingly, cell C_i is split into $m \times m$ equal-sized cells $C_{i1}, C_{i2}, \cdots, C_{im^2}$ and the objects are re-associated with the new cells. In terms of $\mathcal{H}$-grid index, we insert $m \times m$ child cells under cell C_i, and we name C_i as the *parent cell* of cells $C_{i1}, \cdots, C_{im^2}$. Cells without any child cell are named as *leaf cells*, and objects are only associated with leaf cells. Similarly, when objects change their positions, they might need to be moved from one leaf cell to another leaf cell. A cell merge operation will be triggered when the total number of objects associated with any of the leaf cells falls below N_c. All the leaf cells will be removed, and the parent cell then becomes the new leaf cell.

An example $\mathcal{H}$-grid index is depicted in Fig. 2. We assume $N_c = 4$ and $m = 2$. Initially, the entire search space is partitioned into 2×2 equal-sized grid cells, denoted as C_1, C_2, C_3, and C_4. Objects are inserted into corresponding cells. When $|C_1|$ reaches 4, a cell split operation is triggered. C_1 is then partitioned into 2×2 smaller sub-cells, denoted as C_{11}, C_{12}, C_{13}, and C_{14}. C_1 is the parent cell of cells C_{11}, C_{12}, C_{13}, and C_{14}. To facilitate our discussion, we differentiate grid cells based on the level they are. Initial $m \times m$ grid cells are in level 1, their immediate children are in level 2, and so on. The larger the level number is, the smaller the grid cell is. We also plot the index structure in Fig. 2 to facilitate the understanding of the structure of $\mathcal{H}$-grid index. Note all the leaf cells are in grey color in the index.

Compared with traditional grid index, $\mathcal{H}$-grid index is more adaptable to the skewed distribution of objects. In addition, the search expansion in SIMkNN is cell-size based. To be more specific, let R_i represent the original search space and R_{i+1} be the expanded search area. Both R_i and R_{i+1} have rectangular shape and are centered at q. Their side length difference is set to the twice of the side length of the neighbouring cells. Under $\mathcal{H}$-grid index, the size of the neighbouring cells could be very different. If we use the side length of the smallest neighbouring cell to guide the expansion, the expansion will be performed in a slower pace, which helps to expand the search space gradually and hence to increase the number of candidates step by step.

Note that, given a $\mathcal{H}$-grid index with $m = 2$, it can be considered as a form of quad-tree. Hence, the methodology presented in this paper can also work on quad-tree. The advantage of using $\mathcal{H}$-grid index is that flexibly tuning the split factor can help us gain the optimal performance.

2.2 Cell Neighbors

In the following, we introduce a few terms that will be frequently used in the rest of this paper.

For each cell C_i in $\mathcal{H}$-grid index, its *neighbouring cells* locate in eight directions. Based on their relative locations to c_i, those eight neighbouring cells could be partitioned into two clusters, namely *side neighbours* and *angle neighbours*. To be more specific, side neighbours refer to up/down/left/right-side neighbours, and angle neighbours refer to top-left/top-right/bottom-left/bottom-right neighbours. An example is depicted in Fig. 3. Cells labelled as 1, 2, and 5 are up neighbours of the grey colored cell, and hence belong to side neighbours; cells labelled as 6 and 7 are right neighbours, and hence also belong to side neighbours. Note that, a cell might have multiple neighbours in each side direction, but it has one and only one neighbour in each angle direction. The knowledge of its neighbouring cells in eight directions helps the cell to update its neighbours when it performs split or merge action.

Given the fact that multiple neighbouring cells might be located at one side of a given cell c_i in $\mathcal{H}$-grid index, we maintain four containers for cell c_i. Each container of c_i corresponds to one side direction, storing the IDs of c_i's neighbouring cells located at that side direction. Note the neighbouring cells corresponding to each side direction are sorted, based on ascending order of their latitude values or longitude values. For example, the neighbouring cells 1, 2, and 5 shown in Fig. 3 are stored in the order of 1 first, 2 second, and 5 last in the container that represents the up side of the grey cell. These orders can facilitate the computation of the neighbouring cells for the newly split cells.

Last but not least, we would like to define *intersection cells* and *adjacent cells* of any given region R_i. The former

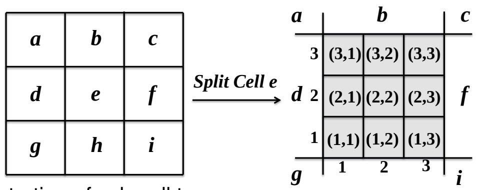


Fig. 4. Illustration of sub-cell types

refers to the grid cells that overlap with R_i, and the latter refers to the grid cells that are adjacent to R_i without any overlap. Both types of cells are useful when implementing the space expansion which will be introduced later.

3 Cell Communication within $\mathcal{H}$-grid

In this section, we propose a *cell communication* technique that makes sure each cell is aware of its neighbouring cells, even in a dynamic environment where cells are expected to perform frequent cell split or cell merge. In the following, we explain the cell communication triggered by cell split and cell merge respectively. Note, only cell split or cell merge actually changes the grid partition of $\mathcal{H}$-grid index.

3.1 Cell communications during splitting

When a cell c_i splits, a set of new sub-cells c_{ij} is generated. We need to inform all c_i's neighbouring cells of c_i's expiration as c_i is no longer available, and hence neighbouring cells of c_i have to replace c_i with proper sub-cells that are newly formed, and meanwhile we also need to find all the neighbouring cells for new sub-cells c_{ij}. We will first explain how to find new neighbouring cells for the newly formed sub-cells, and then explain how to inform c_i's original neighbouring cells of the expiration of c_i and the arrival of new sub-cells.

Given a parameter m, cell c_i is split into $(m \times m)$ sub-cells c_{ij}. Based on the positions of c_{ij} within c_i, they are clustered into three types, namely *angle cells*, *side cells* and *inner cells*. Angle cells refer to the sub-cells that are located at the four corner locations of c_i. There are in total four angle cells, one at each corner, regardless of the value of m. Side cells refer to the cells that are located on up/down/left/right side of the parent cell c_i, and there are in total $4 \times (m-2)$ side cells, among those $m \times m$ sub-cells. Inner cells refer to the rest sub-cells, in total $(m-2)^2$ inner cells. An example split is depicted in Fig. 4. The parent cell e is split into 3×3 sub-cells. Sub-cells $(1,1)$, $(1,3)$, $(3,1)$, and $(3,3)$ are the angle cells, in total four; sub-cells $(1,2)$, $(2,1)$, $(2,3)$, and $(3,2)$ are the side cells, in total $4 \times (m-2) = 4$; sub-cell $(2,2)$ is the inner cell, in total $(m-2)^2 = 1$.

Next, we set forth the algorithm for obtaining neighbour cells for child cells. The main idea here is to apply different strategies to different cells according to their types, positions and the neighbour cell directions. Totally, there are three strategies: (1) *Inheritance Strategy* where the neighbours for the child cell are directly inherited from the neighbours of the parent; (2) *Selective Inheritance Strategy* where a subset of neighbours of the parent are selected as the neighbours for the child cell; (3) *Non-Inheritance Strategy* where no neighbouring cell is inherited from the parent and instead, they can be obtained by simple computation based on the cell positions. In addition, given a parent cell c_i that is partitioned into child cells c_{ij}s, we cluster the neighbour cells of c_{ij}s into two groups based on their sources, namely *out-neighbours* and *in-neighbours*. The former refers to all the neighbouring cells that are the neighbouring cells of the parent cell c_i; and the latter refers to all the neighbouring cells that are newly formed c_{ij}s. The first two strategies work on out-neighbour inheritance and the third strategy focuses on the in-neighbour computation.

Inheritance strategy for angle cells. For a newly generated sub-cell c_{ij} that is an angle cell located at direction x, it shares the same neighbouring cell at direction x as its parent cell c_i. In other words, the neighbouring cell of c_{ij} at direction x could be directly inherited from its parent cell c_i, as both the parent cell c_i and the sub-cell c_{ij} share the same neighbouring cell in the same angle direction x. This direct inheritance is enabled by the fact that there is *only one* angle neighbour in any given angular direction. As shown in Fig. 4, the sub-cell $(3,1)$ is an angle cell of the parent cell e, located in top left direction. Consequently, the neighbouring cell of the sub-cell $(3,1)$ is set to the angle neighbour of e corresponding to the top left direction (i.e., cell a).

Selective inheritance strategy for side cells. This strategy aims to find out-neighbour cells for the sub-cells that are side cells by selectively inheriting the neighbouring cells from the parent. Note this strategy is only applicable for side cells. For the purpose of better illustration, we assume the grid cells are placed in a two-dimensional space. Each cell is represented by (x_s, y_s, x_e, y_e), where x_s and x_e refer to the starting and ending positions of the cell along X dimension respectively, and y_s and y_e refer to the starting and ending positions of the cell along Y dimension respectively. In the following, we consider a cell n that is a neighbouring cell of c_i and another cell c_{ij} that is a side cell newly generated by the splitting of c_i. The details of this strategy are listed below, based on the positions of n and c_{ij}.

- *Case 1.* For newly formed side/angle cells, they inherit the side neighbour cells from their parent that are adjacent to them. To be more specific, if $n.x_e = c_{ij}.x_s \wedge (n.[y_s, y_e] \subseteq c_{ij}.[y_s, y_e] \parallel c_{ij}.[y_s, y_e] \subseteq n.[y_s, y_e])$, n is one of c_{ij}'s left side neighbouring cells; if $n.x_s = c_{ij}.x_e \wedge (n.[y_s, y_e] \subseteq c_{ij}.[y_s, y_e] \parallel c_{ij}.[y_s, y_e] \subseteq n.[y_s, y_e])$, n is one of c_{ij}'s right side neighbouring cells; if $n.y_s = c_{ij}.y_e \wedge (n.[x_s, x_e] \subseteq c_{ij}.[x_s, x_e] \parallel c_{ij}.[x_s, x_e] \subseteq n.[x_s, x_e])$, n is one of c_{ij}'s up side neighbouring cells; if $n.y_e = c_{ij}.y_s \wedge (n.[x_s, x_e] \subseteq c_{ij}.[x_s, x_e] \parallel c_{ij}.[x_s, x_e] \subseteq n.[x_s, x_e])$, n is one of c_{ij}'s down side neighbouring cells.
- *Case 2.* For newly formed angle cells, they inherit certain angle neighbour cells from their parent. To be more specific, given a top-left angle cell c_{ij}, if $c_{ij}.x_e \in [n.x_s, n.x_e] \wedge c_{ij}.y_e = n.y_s$, n is the top-right neighbour cell of c_{ij}; if $c_{ij}.y_s \in [n.y_s, n.y_e] \wedge c_{ij}.x_s = n.x_e$, n is the bottom-left neighbour cell of c_{ij}; given a top-right angle cell c_{ij}, if $c_{ij}.x_s \in [n.x_s, n.x_e] \wedge c_{ij}.y_e = n.y_s$, n is the top-left neighbour cell of c_{ij}; if $c_{ij}.y_s \in [n.y_s, n.y_e] \wedge c_{ij}.x_e = n.x_s$, n is the bottom-right neighbour cell of c_{ij}; given a bottom-left angle cell c_{ij}, if $c_{ij}.y_e \in [n.y_s, n.y_e] \wedge c_{ij}.x_s = n.x_e$, n is the top-left neighbour cell of c_{ij}; if $c_{ij}.x_e \in [n.x_s, n.x_e] \wedge c_{ij}.y_s = n.y_e$, n is the bottom-right

neighbour cell of c_{ij}; given a bottom-right angle cell c_{ij}, if $c_{ij}.y_e \in [n.y_s, n.y_e] \wedge c_{ij}.x_e = n.x_s$, n is the top-right neighbour cell of c_{ij}; if $c_{ij}.x_s \in [n.x_s, n.x_e] \wedge c_{ij}.y_s = n.y_e$, n is the bottom-left neighbour cell of c_{ij}.

- *Case 3*. For newly formed side cells, they inherit certain angle neighbour cells from their parent. To be more specific, given a left side cell c_{ij}, if $c_{ij}.y_e \in [n.y_s, n.y_e] \wedge c_{ij}.x_s = n.x_e$, n is the top-left neighbour cell of c_{ij}; if $c_{ij}.y_s \in [n.y_s, y_e] \wedge c_{ij}.x_s = n.x_e$, n is the bottom-left neighbour cell of c_{ij}; given a right side cell c_{ij}, if $c_{ij}.y_e \in [n.y_s, n.y_e] \wedge c_{ij}.x_e = n.x_s$, n is the top-right neighbour cell of c_{ij}; if $c_{ij}.y_s \in [n.y_s, n.y_e] \wedge c_{ij}.x_e = n.x_s$, n is the bottom-right neighbour cell of c_{ij}; given a up side cell c_{ij}, if $c_{ij}.x_s \in [n.x_s, n.x_e] \wedge c_{ij}.y_e = n.y_s$, n is the top-left neighbour of c_{ij}; if $c_{ij}.x_e \in [n.x_s, n.x_e] \wedge c_{ij}.y_e = n.y_s$, n is the top-right neighbour of c_{ij}; given a down side cell c_{ij}, if $c_{ij}.x_s \in [n.x_s, n.x_e] \wedge c_{ij}.y_s = n.y_e$, n is the bottom-left neighbour cell of c_{ij}; if $c_{ij}.x_e \in [n.x_s, n.x_e] \wedge c_{ij}.y_s = n.y_e$, n is the bottom-right neighbour cell of c_{ij}.

Note that there is no need to compare each parent neighbour cell in the out-neighbour set with the child cell to judge whether it can be inherited. Instead, considering that the neighbouring cells on side directions are sorted in order, as long as we can only obtain the first and the last parent neighbour cells that can satisfy the corresponding judging rule, the cells between them can be directly retrieved.

Non-Inheritance Strategy. Unlike previous strategies where the neighbour cells can be inherited from the parent, this strategy focuses on the in-neighbours, i.e., no inheritance exists since all the in-neighbours refer to the newly generated sub-cells and they can be computed within the scope of the parent cell itself. The neighbouring cell computation is based on the representation of the relative positions for the child cells.

In $\mathcal{H}$-grid index, m is the split factor. Whenever a grid cell splits, $m \times m$ sub-cells will be generated. Each child cell can be identified by (i, j) where i and j are the row index and column index respectively. As show in Fig. 4, cell e is split to 3×3 sub-cells and $(3, 1)$ is the top-left angle-cell, $(1, 3)$ is the bottom-right angle-cell. Using these row and column indices can help us to locate the neighbours for a given child cell. Mainly, there are three cases involved based on the cell type.

- *Case 1: angle-cells.* Given the top-left angle-cell $(m, 1)$, cells $(m-1, 1)$, $(m, 2)$, $(m-1, 2)$ form the down-side, right-side, and bottom-right neighbours of $(m, 1)$. Similarly, we can derive the in-neighbours of three directions for the other three angle cells.
- *Case 2: side-cells.* Given a left side-cell $(x, 1)$ with $x \in [2, m-1]$, cells $(x+1, 1)$, $(x-1, 1)$, $(x, 2)$, $(x+1, 2)$, $(x-1, 2)$ are the up, down, right, top-right and bottom-right neighbour cell respectively. Similarly, we can infer the neighbour cells for other side-cells.
- *Case 3: inner-cells.* Given a child cell that is an inner-cell(x, y) with $x \in [2, m-1] \wedge y \in [2, m-1]$, cells $(x, y+1)$, $(x, y-1)$, $(x-1, y)$, $(x+1, y)$, $(x-1, y+1)$, $(x+1, y+1)$, $(x-1, y-1)$, $(x+1, y-1)$ are the corresponding neighbouring cells in up, down, left, right, top-left, top-right, bottom-left, and bottom-right, respectively.

TABLE 1
Neighbors for child cells in Fig.4 example

Direction	$cell(2,1)$	$cell(3,1)$	$cell(3,2)$	$cell(2,2)$
top-left	d	a	b	$cell(3,1)$
top-right	$cell(3,2)$	b	b	$cell(3,3)$
bottom-left	d	d	$cell(2,1)$	$cell(1,1)$
bottom-right	$cell(1,2)$	$cell(2,2)$	$cell(2,3)$	$cell(1,3)$
left	d	$cell(2,1)$	$cell(3,1)$	$cell(2,1)$
right	$cell(2,2)$	$cell(3,2)$	$cell(3,3)$	$cell(2,3)$
up	$cell(3,1)$	b	b	$cell(3,3)$
down	$cell(1,1)$	$cell(2,1)$	$cell(2,2)$	$cell(1,2)$

Example. To ease the understanding of above strategies, we use the example shown in Fig.4 for illustration. Assume the cell e is split to 3×3 sub-cells and the neighbouring cells for three types of child cells in different directions are summarized in Table 1. For instance, $cell(3, 1)$ is a top-left angle-cell and cell a is its out-neighbour in the top-left direction, which can be obtained by using the *inheritance strategy*. To get its out-neighbours $\{d, d, b, b\}$ in {left, bottom-left, up, top-right} sides, we apply the rules in case 2 of the *selective inheritance strategy*. The case 1 in *non-inheritance strategy* can be applied to $cell(3, 1)$ to compute the in-neighbours $\{cell(3, 2), cell(2, 2), cell(2, 1)\}$ in directions of {right, bottom-right, down} sides. □

After we explain how to find the neighbour cells of newly formed sub-cells, we move our focus to all the cells that have the original cell c_i as one of their neighbour cells, and discuss how to inform them the splitting of c_i. Note that only the out-neighbours of the newly formed angle or side cells (i.e., those neighbour cells identified via inheritance strategy or selective inheritance strategy in above process) are affected in this process. Assume a cell n is an out-neighbour of a newly formed cell c_{ij} at direction X. Cell c_{ij} will compose a message in the form of $\langle ID_r, Dir, ID_p, ID_c \rangle$, where ID_r refers to the cell ID of the receiver (e.g., ID of cell n in our example), Dir refers to the direction in which cell c_i serves as one of the neighbour cells of n (e.g., the opposite of direction X), ID_p refers to the cell ID of the one that splits (e.g., ID of cell c_i in our example), and ID_c refers to the cell ID of a newly formed sub-cell (e.g., c_{ij} in our example). In other words, c_{ij} composes a message $\langle n, opp(X), c_i, c_{ij} \rangle$ to inform cell n that its neighbour cell c_i located at $opp(X)$ direction is no longer valid, and it shall be replaced by new cell c_{ij}.

Example. We continue to use the example depicted in Fig.4 for illustration. When cell e splits, angle-cells $(1, 1)$, $(3, 1)$, $(3, 3)$, $(3, 1)$ and side-cells $(2, 1)$, $(3, 2)$, $(2, 3)$, $(1, 2)$ need to send messages to their out-neighbours. Take cell $(3, 1)$ as an example, since it is the top-left angle-cell, its out-neighbours only exist in five directions, i.e., top-left, left, bottom-left, up and top-right. Based on the results shown in Table 1, the messages that $cell(3, 1)$ sends are listed in Table 2. □

Based on the content of Table 2, we observe that a cell n might receive multiple messages from the same sub-cell c_{ij} because n is an out-neighbour of c_{ij} in multiple directions. This is mainly caused by the fact that cell n is larger than sub-cell c_{ij}. To save the cost of message sending, we propose *message merging* strategy as the solution. In the

TABLE 2
Messages composed by $cell(3,1)$ in Fig.4

Message No.	Out-neighbor Direction	Message Content
1	top-left	{*a, bottom-right, e, (3,1)*}
2	left	{*d, right, e, (3,1)*}
3	bottom-left	{*d, top-right, e, (3,1)*}
4	up	{*b, down, e, (3,1)*}
5	top-right	{*b, bottom-left, e, (3,1)*}

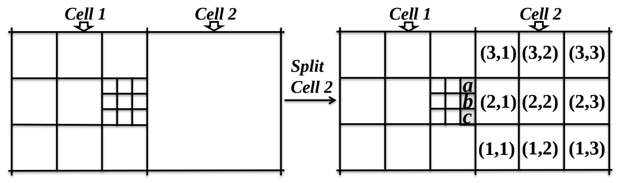


Fig. 5. Example for message translating

case where the size of an out-neighbour n is larger than that of a sub-cell c_{ij}, n is expected to receive multiple messages from c_{ij} where n serves as an out-neighbour in different directions. We merge the messages composed by the same sub-cell with the same receiver together. To be more specific, given two messages $m_1 = \langle ID_{r_1}, X_1, ID_{p_1}, ID_{c_1}\rangle$ and $m_2 = \langle ID_{r_2}, X_2, ID_{p_2}, ID_{c_2}\rangle$, if $ID_{r_1} = ID_{r_2} \wedge ID_{p_1} = ID_{p_2} \wedge ID_{c_1} = ID_{c_2} \wedge X_1 \neq X_2$, we form a merged message m in the form of $\langle ID_{r_1}, X, ID_{p_1}, ID_{c_1}\rangle$. Note direction X might be X_1 or X_2, dependent on the relative position of cell ID_{c_1} to out-neighbour cell ID_{r_1}.

On the contrary, when cell n is smaller than a sub-cell c_{ij}, it is expected that a sub-cell c_{ij} is not only a neighbour cell of n in opposite X direction but also a neighbour cell of n in other directions. An example is shown in Fig. 5. Cell 2 is split into 3×3 sub-cells, and cell b is one of the out-neighbours along left side of $(2,1)$. Accordingly, cell $(2,1)$ will issue a message $\langle b, right, 2, (2,1)\rangle$. However, we notice that $(2,1)$ is not only a neighbour in right direction (the opposite of left direction) but also a neighbour in top-right direction and bottom-right direction. In other words, although cell b is not a neighbour of $(2,1)$ in the top-left direction or in the bottom-left direction, $(2,1)$ needs to send messages $\langle b, \text{top-right}, 2, (2,1)\rangle$ and $\langle b, \text{bottom-right}, 2, (2,1)\rangle$. Accordingly, we propose *message extension* strategy as a solution. When the size of a sub-cell c_{ij} is larger than that of an out-neighbour n that is located at X direction, c_{ij} is a neighbour cell of n in opposite X direction, and it might be a neighbour cell of n in adjacent direction(s) (e.g., top-left and bottom-left are directions adjacent to right; top-left and top-right are adjacent to up). To be more specific, $\langle ID_r, Dir, ID_p, ID_c\rangle$ might be translated to $\langle ID_r, Adj(Dir), ID_p, ID_c\rangle$. Function $Adj(Dir)$ returns the two adjacent directions of Dir. However, examinations have to be performed to verify whether ID_c is a real neighbouring cell in both adjacent directions of Dir, as an extended message is sent only when ID_c is a *real* neighbour cell in that particular direction. For example, cell a is an out-neighbour in the left direction of cell $(2,1)$ and accordingly cell $(2,1)$ will issue a message $m = \langle a, \text{right}, 2, (2,1)\rangle$. $Adj(right)$ returns top-right and bottom-right, but $(2,1)$ is a neighbour of a in bottom-right direction but not in top-right direction. Consequently, message m is only extended for bottom-right direction.

3.2 Cell communications during merging

In $\mathcal{H}$-grid index, when an object is removed from a cell c_{ij}, the total number of objects associated with the parent cell c_i of c_{ij} is decreased by one. When the number falls below the capacity of c_i, a merge of all the sub-cells under c_i is triggered as cell c_i now is able to accommodate all the objects. During cell merging, the neighbour cells of c_{ij} need to update their neighbours as c_{ij}s are no longer valid, and meanwhile we need to find neighbours for cell c_i. In the following, we detail these two steps.

First, cells c_{ij}s need to inform their out-neighbours n that c_{ij}s are no longer available via messages in the form of $\langle n, Dir, c_{ij}, c_i\rangle$, so out-neighbours ns will replace c_{ij}s with c_i. Again, we use the example in Fig. 4 for illustration. Suppose that the 3×3 sub-cells in grey area are going to be merged into a new cell e, and they need to inform their out-neighbour cells their expiration and the arrival of new cell e. For example, cell $(2,1)$ sends following three messages: $\langle d, \text{bottom-right}, (2,1), e\rangle$, $\langle d, \text{top-right}, (2,1), e\rangle$ and $\langle d, \text{right}, (2,1), e\rangle$. Clearly, these messages will be merged to $\langle d, \text{right}, (2,1), e\rangle$ since cell $(2,1)$ is only on the right side of cell d.

Second, we need to find neighbours for the newly formed cell c_i by leveraging the out-neighbours of cells c_{ij}s. Particularly, the neighbour cells of c_i in the angle direction can be inherited directly from angle cells c_{ij}s that are located at $(1,1)$, $(1,m)$, $(m,1)$, and (m,m). For example, as shown in Fig. 4, the top-left neighbour of cell e is the same as the top-left neighbour of angle cell (3.1) (i.e., cell a), the top-right neighbour of cell e is the same as the top-right neighbour of angle cell $(3,3)$ (i.e., cell c), and so on. The neighbours in side directions for the merged cell c_i are the union of the out-neighbours of the side-cells before merging.

4 SIMkNN Query Processing

$\mathcal{H}$-grid index can improve the efficiency of *k*NN query processing, since it is able to obtain an appropriate search scope for moving objects and the partial road network, especially when the objects are in skewed distribution. In this section, we firstly explain the overall query processing framework for SIMkNN and then present the implementation details for the critical step, i.e., $\mathcal{H}$-grid based *Result Space Estimation*.

4.1 Overall query processing framework

SIMkNN adopts a $\mathcal{H}$-grid index to partition the entire search space that captures all the positions of the moving objects at time t into a hierarchical grid of square-shaped cells with different levels and sizes. Each leaf cell maintains a list of objects whose coordinates fall inside the cell space; while each parent cell C_i maintains a list of $m \times m$ child cells that are generated by splitting C_i. Besides, each cell is aware of its neighbors in eight directions at any time. R-tree in SIMkNN is used to store the connectivity information, such as edges and vertices, of the road network where the objects move on. Then, the snapshot *k*NN query processing in SIMkNN consists of two steps, namely *result space estimation* and *kNN retrieval*.

To be more specific, SIMkNN first locates the query point q into a leaf cell c_{ij}, and slowly expands the search space based on the minimum side length of any neighbouring cells of the current search space, until top-k objects that are closest to q based on Euclidean distance are located. To be more specific, let $N(c_{ij})$ refer to the set of neighbouring cells, and let d refer to the minimum side length of any neighbouring cells, i.e., $d = min_{\forall c \in N(c_{ij})} c.side$. Each expansion expands the search space from one square R_i to another larger square R_{i+1} with $R_{i+1}.side - R_i.side = 2d$. Once we locate the set of k objects having smallest Euclidean distances to q, we complete the step of result space estimation, and we start the next step.

In the kNN objects retrieval step, SIMkNN utilizes the estimated area resulted from previous step to launch a range query, extracting the edges and vertices from the R-tree and re-constructing a partial road network that covers the estimated search area. We adopt an approach that is similar as *INE (incremental network expansion)* [16] to expand the network and to derive the network distance, and each vertex in the partial network locally constructed maintains a network distance from itself to the query point q. A local result set Res is maintained to preserve the top-k objects with minimum network distances to q, and it is guaranteed that at least k objects are within the estimated area. Let V_{exp} refer to the set of vertices locally available with at least one adjacent vertex that is not locally available. The search can be safely terminated if the maximum network distance from a local result to q does not exceed the minimum distance between any vertex in V_{exp} and q, i.e., if $max_{\forall o \in Res}||o, q|| \leq min_{\forall v \in V_{exp}}||v, q||$[1], Res is confirmed to be the final result set and the search can be terminated. The correctness of the above mentioned kNN query processing under SIMkNN is guaranteed by Theorem 1.

Theorem 1. The proposed kNN query processing of SIMkNN is correct, i.e., the result set Res returned by SIMkNN is guaranteed to be the real result set.

Proof. Assume the above statement is not correct, and there is at least one object $o' \notin Res$ that has a shorter network distance to q than a result object $o \in Res$, i.e., $\exists o' \notin Res \wedge \exists o \in Res, ||o', q|| < ||o, q||$. Here, q refers to the query point. Because $o' \notin Res$, object o' has not yet been visited by our kNN objects retrieval step. This means, assume o' is located on edge (v_a, v_b) of the road network, at least one vertex of the edge has not yet been reached. To facilitate our discussion, we introduce two notations V_{exp}, and V_{not}. The former V_{exp} refers to the set of vertices locally available with at least one adjacent vertex that is not locally available, and the latter V_{not} refers to the set of vertices that are not locally available. Obviously, the fact that o' has not yet been visited implies following three cases, i) $v_a \in V_{exp} \wedge v_b \in V_{not}$; ii) $v_a \in V_{not} \wedge v_b \in V_{exp}$; and iii) $v_a \in V_{not} \wedge v_b \in V_{not}$. Since any path from a vertex $\in V_{not}$ to q has to bypass at least one vertex in V_{exp}, we have $\forall v \in V_{not}$, $||v, q|| > min_{\forall v_e \in V_{exp}}||v_e, q||$. Together with the termination condition of kNN retrieval (i.e., $max_{\forall o \in Res}||o, q|| \leq min_{\forall v \in V_{exp}}||v, q||$), we have $||v_a, q|| \geq max_{\forall o \in Res}||o, q||$ and $||v_b, q|| \geq max_{\forall o \in Res}||o, q||$. On the other hand, because o' is located on the edge (v_a, v_b), we have $||o', q|| \geq min(||q, v_a||, ||q, v_b||)$. As a result, $||o', q|| \geq min(||q, v_a||, ||q, v_b||) \geq max_{\forall o \in Res}||o, q|| \geq ||o, q||$ which contradicts our assumption. Consequently, our assumption is invalid and the proof completes. ■

1. Notation $||o, o'||$ refers to the network distance between two objects o and o'.

For the step of kNN objects retrieval, many existing kNN search algorithms are applicable and its implementation is not the main contribution of our work. Consequently, we only focus on the implementation of result space estimation step, whose details will be presented in next section.

4.2 $\mathcal{H}$-grid based Result Space Estimation

The size of the estimated search space has an impact on the range query issued in the step of kNN objects retrieval. The larger the estimated search space, the larger the local network that has to be searched by the range query. Consequently, we adopt a conservative approach to slowly expand the search space centered at the query point q until the search space covers at least k objects. Because the search space is indexed by $\mathcal{H}$-grid index, the neighbouring cells surrounding the current search space are very likely to be of different size, which enables gradual expansion of the search space. In the following, we mainly explain one key building-block of space expansion, that is *how to decide the expansion size*.

To simplify our discussion, we first introduce a few of data structures used throughout the result space estimation. Given a search space, set CS_{neig} refers to all the neighbouring cells of the current search space. A cell C_i is a neighbouring cell of a given search space SS iff SS overlaps with C_i or C_i is adjacent to SS. When the current search space is expanded to a larger search space SS', the set of original neighbouring cells is partitioned into two disjoint subsets, $CS_{overlap}$ and CS_{inside}. The former refers to the set of cells that intersect or overlap with the expanded search space SS'; the latter refers to the rest of the cells that are fully covered by the expanded search space SS'. Set $CS_{covered}$ refers to the set of cells that are fully covered by the current search space.

We determine the expansion size based on the size of all the neighbouring cells, i.e., those cells included in set CS_{neig}. Since we adopt a conservative approach, we set the expansion size to the minimum side length of a neighbouring size, i.e., $\text{MIN}_{\forall c \in CS_{neig}} c.side$. Now the problem of deciding the expansion size is converted to finding the set of neighbouring cells of a search space. With the help of $CS_{covered}$, CS_{inside}, and $CS_{overlap}$, the new set of neighbouring cells of the expanded search space can be derived based on Equation (1). Note, function $\text{NEIG}(c)$ is to retrieve all the neighbouring cells in eight directions of cell c.

$$NS = (\bigcup_{c \in CS_{inside}} \text{NEIG}(c) - CS_{covered}) \cup CS_{overlap} \quad (1)$$

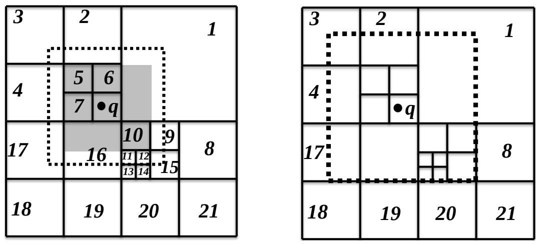
Algorithm 1 lists the pseudo code of the search space expansion process. It first locates the center cell, the cell where the query point q is located, and initializes the search space to the center cell (lines 1-2). CS_{neig} is empty and $CS_{covered}$ only contains the center cell (lines 3-4). It then starts the expansion process. The expansion continues if there are less than k objects within the current search space, i.e., the total number of objects that are associated with any cell included in $CS_{covered}$ is smaller than k (line 5).

Algorithm 1: Space Exp. for Estimated Result Space

```
Input  : Query point $q$; number of objects $k$; $\mathcal{H}$-grid index;
Output: Estimated result space after progressive expansion: searchSpace
1  $centerCell \leftarrow$ the cell that contains query point $q$;
2  $searchSpace \leftarrow centerCell$;
3  $CS_{covered} \leftarrow centerCell$;
4  $CS_{neig} \leftarrow \emptyset$;
5  while there are in total less than $k$ objects in $searchSpace$ do
6      $CS_{inside} \leftarrow \emptyset$;
7      if $CS_{neig}$ is empty then
8          $CS_{neig} \leftarrow$ NEIG($centerCell$);
9      else
10         for each cell $c \in CS_{neig}$ do
11             if $c$ is fully covered by $searchSpace$ then
12                 $CS_{inside} \leftarrow CS_{inside} \cup \{c\}$;
13         $CS_{overlap} \leftarrow CS_{neig} - CS_{inside}$;
14         $CS_{covered} \leftarrow CS_{covered} \cup CS_{inside}$;
15         $NS \leftarrow \bigcup_{c \in CS_{inside}}$ NEIG($c$);
16         $CS_{neig} \leftarrow (NS - CS_{covered}) \cup CS_{overlap}$;
17     $expSize \leftarrow$ MIN$_{\forall c \in CS_{neig}} c.side$;
18     $searchSpace \leftarrow$ EXP($searchSpace$, $expSize$);
19 return searchSpace
```

It empties set CS_{inside} that captures all the neighbouring cells of the current search space but will be fully covered by the expanded search space after expansion. It then locates the neighbouring cells that will be maintained by CS_{neig} set. Note that for the first expansion, CS_{neig} is empty and we invoke function NEIG($centerCell$) to initialize CS_{neig} to the set of neighbouring cells of the cell that query point q is located (lines 7-8). For all the following expansions, CS_{neig} maintains the set of neighbouring cells of the current search space before expansion. We need to update CS_{neig} for the expanded search area. We scan cells in CS_{neig} and maintain cells that are fully covered by the expanded search space in CS_{inside} (lines 10-12), and the rest in $CS_{overlap}$ (line 13). Next, $CS_{covered}$ is updated by including cells in CS_{inside} (line 14). We derive the new set of neighbouring cells based on Equation (1) (line 15). CS_{neig} is then updated by excluding those fully covered cells (i.e., cells in $CS_{covered}$) from NS and including those still overlapped cells (i.e., cells in $CS_{overlap}$) (line 16). The expansion size is set to the minimum side length of any neighbouring cell maintained by CS_{neig} (line 17), and the search space is then expanded to complete one expansion. The above process continues until the number of objects inside the current search space reaches k.

Example. As shown in Fig. 6(a), initially $centerCell$ is set to the cell where q locates. During the first expansion, CS_{neig} is set to the set of neighbouring cells of $centerCell$, i.e., $CS_{neig} = \{1, 5, 6, 7, 10, 16\}$, and the size of expansion, i.e., $expSize$ is set to the side length of the smallest neighbouring cell that is cell 5 or cell 6 or cell 7 or cell 10. The grey coloured region represents the search space after the first expansion. During the second expansion, neighbouring cells 5, 6, 7, 10 become fully covered and hence $CS_{covered} = CS_{inside} = \{5, 6, 7, 10\}$, and $CS_{overlap} = \{1, 16\}$. $CS_{neig} = \{1, 2, 3, 4, 9, 11, 12, 15, 16, 17\}$. The expansion size is then set to the side length of the smallest neighbouring cell (i.e., cell 11 or cell 12), and the dotted-



(a) 1st and 2nd expansion (b) 3rd expansion

Fig. 6. A demonstration for result space estimation

line in Fig. 6(a) bounds the expanded search space after the second expansion. During the third expansion, cell 11 and cell 12 are fully covered, with $CS_{inside} = \{11, 12\}$ and $CS_{covered} = \{5, 6, 7, 10, 11, 12\}$. The new set of neighbouring cells $CS_{neig} = \{1, 2, 3, 4, 9, 13, 14, 15, 16, 17\}$, and expand size is set to the side length of the smallest neighbouring cell (i.e., cell 13 or cell 14). The square region bounded by the dotted line in Fig. 6(b) represents the search space after the third expansion. □

5 Experiments

In order to evaluate the performance of $\mathcal{H}$-grid based kNN query for moving objects in road network, we have performed a comprehensive simulation, from following three perspectives. i) We evaluate the performance of $\mathcal{H}$-grid index embedded with the cell communication in terms of the memory usage and update time, as reported in Section 5.1; ii) we study the effects of result space estimation in SIMkNN, including the number of objects that are contained in the estimated space and its time efficiency, as reported in Section 5.2; iii) we investigate the overall performance of SIMkNN query processing, as reported in Section 5.3. For the purpose of comparison, we implemented two baselines. One is an in-memory version of MOVNet based on traditional grid index. The other is IER-PHL [17], which is the state-of-the-art in-memory kNN algorithm. Note that, all algorithms are implemented using Java with JDK 1.7.0.

All experiments in this section are based on a mixture of real and synthetic data sets. We use the road network of Los Angles, CA, USA, containing 264,521 edges and 193,320 nodes. The moving objects on the road network are generated using Brinkhoff road network generator [18]. Totally, we simulate the scale of 1 million moving objects, whose scale is much larger than previous studies. Fig. 7 shows the data distribution at time t. Each square denotes a small area and the intensity of the color indicates the density of the moving objects. For each set of experiments, we generate $1,000$ random queries, and report the average performance. All experiments are evaluated on a PC with Intel(R) Xeon(R) CPU E5-2637 3.50GHz processor and 8 GB RAM with Ubuntu Linux 14.04.

5.1 Index Performance

In our first set of experiments, we evaluate the performance of $\mathcal{H}$-grid index, as compared with grid index employed by MOVNet. The memory usage and the update efficiency are employed as the main performance metrics.

Memory usage. We report the memory overhead of $\mathcal{H}$-grid as a function of cell capacity c and split factor m in

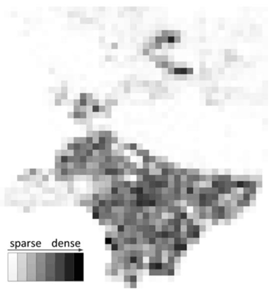


Fig. 7. The distribution of objects on Los Angles road network

Fig. 8(a) and Fig. 8(b) respectively. Note that, cell capacities of the grid index are represented by the average number of moving objects in each cell. Three main observations made from the simulation results are listed below. First, $\mathcal{H}$-grid takes up more memory space for a given cell capacity c, as compared with grid index. This is because $\mathcal{H}$-grid index is more complicated and it needs extra memory space to maintain the hierarchical structures. Second, the memory usage of $\mathcal{H}$-grid index varies, as c or m changes. We can see from Fig. 8(a), memory usage of $\mathcal{H}$-grid increases as cell capacity increases from 35 to 40. The main reason for this is because the inner data structure we used for storing the objects of each cell is an array based list. Specifically, array based list initially needs to be allocated with a fixed maximum size max_size, when the number of objects exceeds max_size, JVM will dynamically increase it to a much bigger one which then causes the memory overhead and produces the results shown in Fig. 8(a). Third, though the grid index shows a stable and relatively small memory usage among the test cases in Fig. 8(a), it does not represent that the grid index consumes stable memory for all cases. On the contrary, as shown in Fig. 8(c), the grid index could be very memory consuming if we increase its cell number.

Update time. The update operation is triggered when an object moves from one cell to another. Fig. 8(d) and Fig. 8(e) report the update time of grid index and $\mathcal{H}$-grid index with regard to the cell capacity c and split factor m, respectively. In general, both indices incur short update time, e.g., , the update time for all test cases shown in Fig. 8(d) and 8(e) is below 0.035 ms. Compared with the grid index, $\mathcal{H}$-grid index requires slightly longer update time since it incurs split or merge operations while grid index does not. Hence, as shown in Fig. 8(d), curves corresponding to two $\mathcal{H}$-grid indices are above that of the grid index. Besides, the grid index has a stable update time with respect to the cell capacity, however, the update time for $\mathcal{H}$-grid index has an obvious descending trend when the cell capacity grows. The reason behind is that a larger cell capacity indicates a larger cell range, which can reduce the possibility of update operation, namely the object has a smaller chance to move from one cell to another. The last observation made from Fig. 8(e) is that the update time of $\mathcal{H}$-grid index decreases as the split factor increases and this trend is more obvious when the cell capacity is small. This can be explained by the reason that a larger split factor can help $\mathcal{H}$-grid maintain more information in the same level of cells and the update operations probably only happen in a very limited area within the same cell level. On the other hand, a smaller split factor generates a $\mathcal{H}$-grid with more levels and it is very likely that the split and merge operations are done across multiple levels which incur many recursive processing steps.

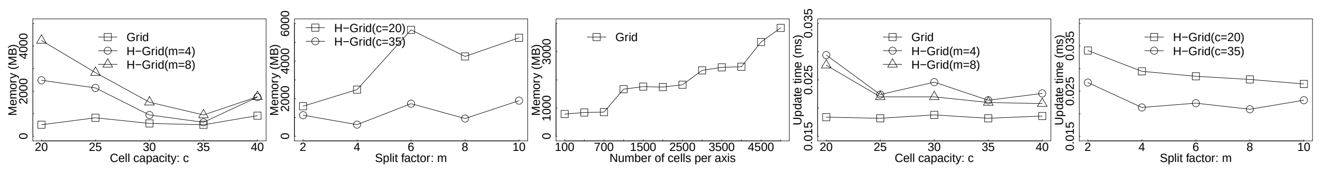
In conclusion, though the grid index has quick update, it gains its time efficiency by sacrificing the space resource, i.e., the memory. Hence, the performance of MOVNet is also limited. The $\mathcal{H}$-grid index requires slightly longer update time as compared with the grid index, but they two are in the same order of magnitude. More importantly, $\mathcal{H}$-grid index scales well with the memory through tuning the parameters of the cell capacity and split factor. As a result, SIMkNN could be more scalable when performing in-memory kNN query processing. Next, we further investigate the performance of SIMkNN.

5.2 Result Space Estimation Effect

In this section, we compare the effect of result space estimation in both MOVNet and SIMkNN. Specifically, two metrics were used for the evaluation: (1) the number of objects included in the estimated space and (2) the time spent on estimation.

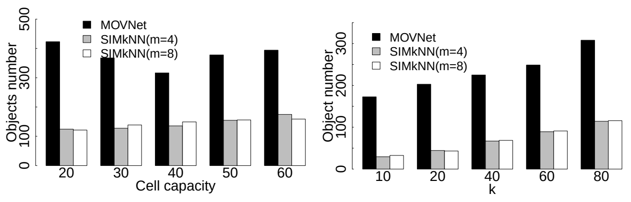
Object number. Fig. 9(a) and Fig. 9(b) plot the number of objects that are evaluated in SIMkNN under $\mathcal{H}$-grid index and that in MOVNet under the grid index. There are three main observations. First, the estimated result space of MOVNet bounds more objects than that of SIMkNN, which justifies that estimated space of SIMkNN under $\mathcal{H}$-grid index is more effective as it covers a smaller search space. For example, the number of objects evaluated under MOVNet is twice of that under SIMkNN, as shown in Fig. 9(a). Second, the object number increases as the cell capacity grows, e.g., for SIMkNN with $m = 8$, its object number in the test case of cell capacity 20 is 121.37 and it increases to 158.8 when cell capacity grows to 60. Third, as shown in Fig. 9(b), the number of objects grows proportionally with the query request k for both MOVNet and SIMkNN. This is because the larger the k is, the larger the search space is. In addition, the number of objects fallen within the search space of SIMkNN is far less than that of MOVNet, e.g., for test case of $k = 10$, only 24.64 objects in average are retrieved for the network distance based processing in SIMkNN while MOVNet needs to consider 172.69 objects. This gap is narrowed as k grows. Therefore, compared with MOVNet, it is reasonable to infer that the smaller the k is, the more significant the advantage of SIMkNN is.

Estimation time. Fig. 10 reports the time needed for result space estimation in both MOVNet and SIMkNN as well as their corresponding ratios with respect to the overall query processing time. Clearly, for most test cases, SIMkNN consumes more time than MOVNet for estimating the result space. This is because the space expansion in SIMkNN involves the computation overhead of finding the neighbors for search area. As a result, SIMkNN needs more time than MOVNet for the result space estimation. There is one exceptional case in each figure. As shown in Fig. 10(a), when the cell capacity is 60, the estimation time of both SIMkNN bars has decreased significantly whereas the MOVNet bar grows longer, which leads to the result that SIMkNN spends less time than MOVNet to complete the estimation when the cell capacity is 60. This is interesting since SIMkNN involves more computation overhead than MOVNet and



(a) Memory usage vs. c (b) Memory usage vs. m (c) Memory usage of grid (d) Update time vs. c (e) Update time vs. m

Fig. 8. Index performance study



(a) Object number ($k = 100$) (b) Object number ($c = 30$)

Fig. 9. The Object number of estimation

it is expected to consume more time. This contradiction is caused by the fact that as cell capacity grows, MOVNet may need to retrieve more objects, and the time for doing this cancels out the overhead of SIMkNN or even makes the MOVNet less efficient. The other exceptional case refers to the test case of $k = 10$ in Fig. 10(b). Given a large split factor and a small k, SIMkNN may not need as much computation as MOVNet does to expand the space. This is because that MOVNet is not adaptable to the skewed distribution of objects, so more expansions and more objects may require extra computation effort from MOVNet.

Last but not least, though the time required by the search space estimation step in SIMkNN is generally more than that in MOVNet, the object number in the estimated area of SIMkNN is much smaller than that of MOVNet. In other words, the input to the next step of network distance based evaluation of SIMkNN is much smaller than that under MOVNet. As shown in Fig. 10(c) and Fig. 10(d), the estimation time only account for less than 1% of total query time in all cases though SIMkNN has a little bit higher ratio than that of MOVNet. Note that the ratio gap between them decreases as the cell capacity or k grows in general. Thus, the inefficiency so far for SIMkNN is not a drawback and instead, the effort for returning less objects will help SIMkNN save computation in the next phase that is much more expensive.

5.3 kNN Query Performance

In this set of experiments, we study the kNN query performance of SIMkNN in terms of the response time. First, to demonstrate the efficiency of the in-memory index of SIMkNN, we implemented its disk version Disk_HgkNN, where the road network data indexed by the R-tree is stored in the disk while the $\mathcal{H}$-grid for indexing the moving objects is maintained in the memory. Second, we compare the performance of SIMkNN with MOVNet and another state-of-the-art kNN algorithm IER-PHL [17] that is originally designed for static objects. To facilitate the comparison and to demonstrate the impact of location updates on the performance of kNN searches, we simulate different movement scenarios where moving objects have different location update frequencies. Third, we study the relation between response time and space requirements in terms of memory usage for different algorithms. Last but not least, we report

TABLE 3
Parameters settings under different memory usages

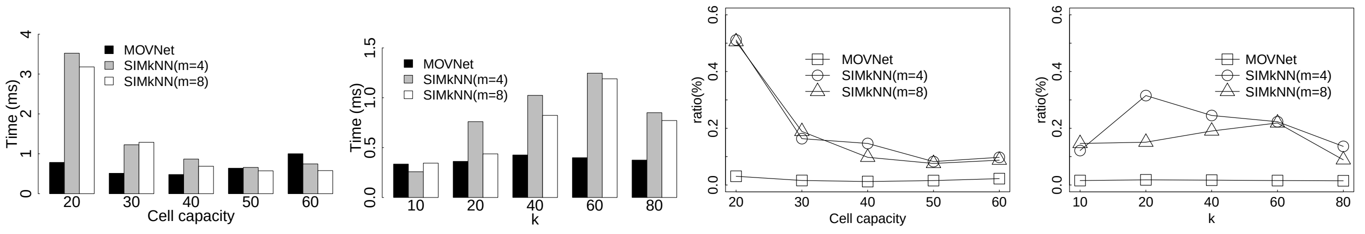
	0.8G	1.0G	1.2G	1.6G	1.8G
SIMkNN	m=2, c =100	m=4, c =150	m=8, c=125	m=6, c=200	m=2, c=75
MOVNet	c =100	c = 2.4	c = 1.7	c =1	c = 0.16
IER-PHL	c=63	c=59	c=56	c=50	c=48

the number of objects that need evaluating during the query processing under different algorithms to explain the main factor that contributes to the high performance of SIMkNN.

Memory vs Disk. Fig. 11 plots the query processing response time of SIMkNN and its disk version Disk_HgkNN as a function of k, cell capacity c and the split factor m, respectively. Moreover, corresponding I/O counts for Disk_HgkNN are also plotted. First of all, compared with SIMkNN, Disk_HgkNN consumes much more time for all cases. This is because that retrieving the topological structure information of the road network from the disk requires many I/O operations (in thousands), as demonstrated in Fig. 11(b), Fig. 11(d) and Fig. 11(f) accordingly. Second, as shown in Fig 11(a), the response time of both SIMkNN and Disk_HgkNN grows proportionally with k, which is obvious since bigger k requires more query processing work to be done. Moreover, the gap between them becomes larger after $k = 80$. The reason for this is that Disk_HgkNN requires more I/O operations to retrieve more road network data from the disk as k grows, which is shown in Fig. 11(b). Third, for cell capacity and split factor, the curves in Fig. 11(c) and Fig. 11(e) fluctuate and the optimal case exists for different parameter settings. Generally, based on the results shown in the figure, SIMkNN can report the answer for $k = 10$ from one million objects within 0.2 seconds, which is twice faster than its disk version Disk_HgkNN. Hence, placing both R-tree and $\mathcal{H}$-grid in the memory can significantly reduce the response time.

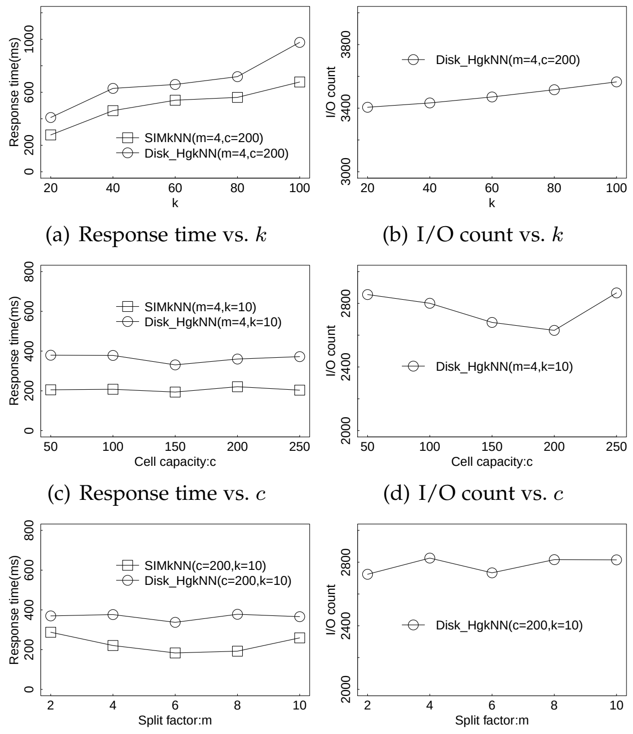
Last but not least, we also notice an interesting trend from Fig. 11(d) that the I/O count firstly decreases as cell capacity c increases from 50 to 200, and then suddenly grows up when $c = 250$. This can be explained by the reason that with the increment of cell capacity, the estimation space will shrink so that Disk_HgkNN retrieves less road network data from the disk. However, when c increases to 250, the cells are relatively coarse-grained so that merely one expansion would require much road network information. This finding also tells us that carefully tuning the parameter c is important to derive the desired efficiency. The similar result is also applicable to the split factor m.

Movement Scenarios. In this part, we simulate different movement scenarios by controlling the number of objects changing their positions. 0% movement indicates that there is no object moving, while 100% movement means that all the objects are moving and have their locations changed. In addition, we fix the space requirement (0.8GB) by em-



(a) Estimation time ($k = 100$) (b) Estimation time ($c = 30$) (c) Estimation time ratio ($k = 100$) (d) Estimation time ratio ($c = 30$)

Fig. 10. The performance of result space estimation



(a) Response time vs. k (b) I/O count vs. k

(c) Response time vs. c (d) I/O count vs. c

(e) Response time vs. m (f) I/O count vs. m

Fig. 11. The performance of kNN query processing (memory vs disk)

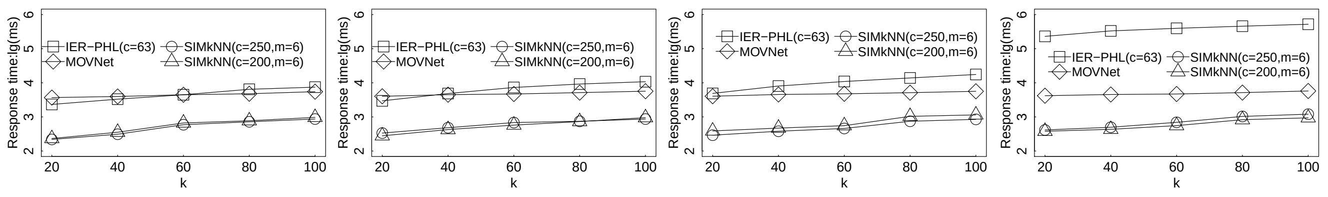
pirically determining the parameters of each algorithm. In order to avoid skewness towards large values of long response time, Fig. 12 is plotted with a logarithmic scale of base 10. Note that, the response time here consists of index update time and kNN query processing time. As we can see from Fig. 12, SIMkNN outperforms the state-of-the-art method IER-PHL (more than an order of magnitude) and MOVNet consistently under all the cases. Specifically, what we found interesting here is that, even for the static case where no position update occurs, e.g., Fig.12(a), our SIMkNN is still faster than IER-PHL. The reason behind this could be explained by: 1) to obtain k-nearest objects by Euclidean distance, IER-PHL needs to retrieve many R-tree nodes, which is very time-consuming, and 2) when k value is big (e.g. 100), it costs much time for IER-PHL to obtain subsequent Euclidean neighbors, because IER-PHL needs to repeatedly find the next Euclidean nearest object until its network distance is greater than the network distance of current k-th candidate object. However, as for SIMkNN, since k is relatively small as compared with the number of total moving objects (1 million in our experiment), it can obtain k-nearest neighbors quickly by expanding a small portion of the road network.

Furthermore, when comparing MOVNet with SIMkNN, we can see from all figures that MOVNet needs more time for answering the kNN query than SIMkNN under different memory usages. Specifically, as shown in Fig. 12 where MOVNet and SIMkNN both consume 0.8GB memory, SIMkNN is faster than MOVNet for almost one order of magnitude in terms of response time. The second observation is that the response time for both methods experiences a rise with respect to k since more space expansions and network distance based evaluations are needed.

Space Requirements. In next set of experiments, we study the efficiency of different algorithms under different memory space requirements. To this end, we performed extensive experiments to empirically determine the relations between the parameters and the memory usages. The results are shown in Table 3. The response time for kNN query ($k = 10$) in Fig 13 is plotted with a logarithmic scale of base 10. There are two main observations. First, SIMkNN consistently outperforms other algorithms under different memory consumption. Specifically, it is one order of magnitude faster than IER-PHL, while 1.2 orders of magnitude faster than MOVnet at the beginning and the gap between MOVNet and SIMkNN narrows as the memory usage increases, e.g., they are almost same when memory usage is 1.8GB. The reason for the narrowing gap is that each cell in the grid index of MOVNet is shrinking which makes the overhead of query processing within MOVNet smaller, i.e., less objects are retrieved after space estimation. Second, both SIMkNN and IER-PHL are relatively insensitive to the varying memory usage. For SIMkNN, it is possibly because the effect of one factor (e.g., split factor) cancels out that of the other one (e.g., cell capacity). As for IER-PHL, the R-tree index for objects plays an important role to make it stable.

Computed object number. We have already demonstrated the excellent performance of SIMkNN in above experiments. In the following, we report the number of objects that require evaluation during kNN search, a key factor that affects the search performance. To be more specific, we report the number of objects ($|O_{in}(SS)|$) derived from estimated search space, and the total number of objects ($|O_{evl}|$) that have been evaluated during the query processing of SIMkNN and MOVNet. Note that we exclude IER-PHL from this set of experiments because of its poor performance. We also report the ratio $\frac{|O_{in}(SS)|}{|O_{evl}|}$. If $\frac{|O_{in}(SS)|}{|O_{evl}|} > 1$, the extra space expansion is needed; otherwise the final query results can be found within the estimated result space. The higher the ratio value is, the more the extra effort is required.

Fig. 14(a) reports the $|O_{evl}|$ number for test cases depicted in Fig. 12. When restricting the memory usage to 0.8GB, the number of computed objects in SIMkNN only accounts for maximum 20% of the object number under



(a) 0%movement (b) 25%movement (c) 50%movement (d) 100%movement

Fig. 12. The performance of kNN query processing in different movement scenarios(0.8G)

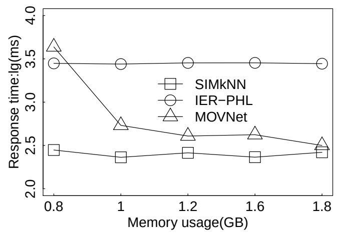


Fig. 13. Response time varying memory usage (k=10)

MOVNet, which well explains why SIMkNN outperforms MOVNet with a much shorter response time as shown in Fig. 12. In addition, from Fig. 14(b) we can see that the value of $\frac{|O_{in}(SS)|}{|O_{evl}|}$ under MOVNet is greater than 2 and it increases slightly as k grows, which indicates that objects in estimated area of MOVNet cannot satisfy the query request and extra space expansions after the estimation phase are needed. On the contrary, the ratio values under SIMkNN curves are below 1 which demonstrates that the expected query results fall inside the estimated area and no more extra expansion is needed. Fig. 14(c) and Fig. 14(d) report the results under 1.6GB memory usage. Note that, even two series of bars for SIMkNN have same number of objects at $k = 80$ as shown in Fig. 14(c), SIMkNN with cell capacity of 100 is faster than that with cell capacity of 50. This is because, in order to find $k = 80$ NNs, SIMkNN of $c = 50$ needs to expand the estimated area derived from the initial step of kNN search, which imposes overhead to the response time. The ratio values shown in Fig. 14(d) prove this explanation.

6 Related Work

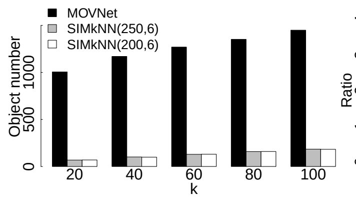
The problem of kNN query processing in spatial database has been well studied. A large number of methods could be found in literature for kNN search. We group them to four categories based on the distance metric employed and whether the object is static.The work in first two categories focuses on the objects that are static, while we are interested in kNN search over moving objects. In the end of this section, we also present some related work for in-memory moving objects indexing.

Euclidean Distance based *k*NN Query for Static Objects. Many works in this category use R-tree [19] to answer nearest neighbor queries. Roussopoulos et al. [20] proposed an efficient branch-and-bound R-tree traversal algorithm for searching kNNs, where they introduced two metrics to guide an ordered depth-first spatial search. Hjaltason et al. [21] described an incremental kNN algorithm which reduces the amount of R-tree nodes accessed, disk I/O as well as the number of distance calculations. There are also other tree based kNN algorithms. For example, VoR-Tree [22] incorporates Voronoi diagram into an R-tree and benefits from both the neighborhood exploration capability of Voronoi diagrams and the hierarchical structure of R-trees.

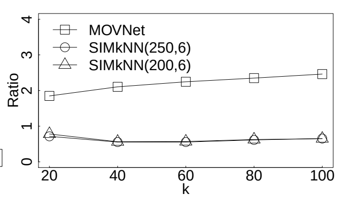
Network Distance based *k*NN Query for Static Objects. Lee et al. presented *ROAD* [23] to precompute the network distance within a sub-network derived from recursively partitioning the original network, then the sub-networks that do not contain any object will be skipped from examination. The pruning effect of *ROAD* is not effective when the objects are widely scattered, and to address this, a balance tree G-tree [24] was proposed and it outperforms *ROAD* by only accessing tree branches containing objects. Deniel et al. [25] designed a unified framework by emphasizing the acceleration of shortest path processing for points-of-interest related queries and the online version of their algorithm outperformed both G-tree and *ROAD*. More recently, Tenindra et al. [17] revisited *Incremental Euclidean Restriction* (IER) method and found it is often the best performing technique if a simple improvement is added. The techniques involved in this category are different from ours since they are aimed for static objects where the frequency of objects location updates is very low. However, location update is very frequent in our work, as we focus on objects that keep moving in road network.

Euclidean Distance based *k*NN Query for Moving Objects. Yu et al. [15] presented a hierarchical grid model as well as an algorithm for processing kNN query over moving objects. Though our work is based on the idea of hierarchical grid model, we improved the model with the novel cell communication technique and integrated it with the dual-index driven kNN algorithm. Recently, to handle the vast volume of data and concurrent queries that are increasingly common in kNN applications, Yu et al. [26] designed a dynamic strip index (DSI) for distributed computing and based on which, a distributed kNN search algorithm was also proposed. All the above works in this category do not consider network distance computation, and hence they are *not* suitable for our problem setting.

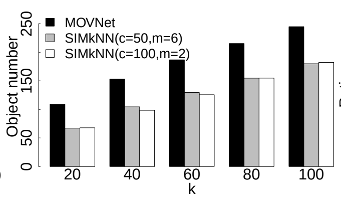
Network Distance based *k*NN Query for Moving Objects. Our problem belongs to this category. Wang et al. [6], [8] first proposed a dual-index driven system called MOVNet which can deal with kNN queries. SIMkNN improves the kNN query processing within MOVNet as follows: (1) SIMkNN can better adapt to the skewed distribution of moving objects; and (2) SIMkNN outperforms MOVNet in terms of time and memory efficiency. In addition, MOVNet maintains the connectivity information of road network in an on-disk R-tree, whereas SIMkNN is a full in-memory kNN search technique. Chen et al. [27] proposed a method that can reduce the updates via a dynamic data structure *adaptive unit* to capture the movement bounds of the objects. Though



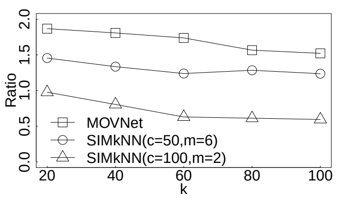
(a) Computed object number $|O_{evl}|$ (0.8GB)



(b) $\frac{|O_{in}(SS)|}{|O_{evl}|}$ (0.8GB)



(c) Computed object number $|O_{evl}|$ (1.6GB)



(d) $\frac{|O_{in}(SS)|}{|O_{evl}|}$ (1.6GB)

Fig. 14. Insights study with different memory usage

that technique can be extended to support *k*NN query, its performance depends on the prediction method used for update. Moreover, the factor of data distribution is also not considered. Lee et al. [28] proposed a lightweight spatial index based on a hierarchical spatial data structure called *geohash* [29], which is a geocoding system for latitude and longitude. They demonstrated the implementation of the index on top of HBase to support the spatial query processing. Their method is based on a distributed implementation while SIMkNN is centralized.

Finally, in terms of **in-memory indexing for the moving objects**, managing a very large number of location updates is a critical issue that needs to be carefully addressed. Frentzos [30] proposed the FNR-Tree which consists of two layers of R-Trees, the lower layer is a 2-D R-Tree to index the network structure while on top of it is a forest of 1-D R-Trees indexing the moving objects inside a given link of the network. Due to the reason that each edge in the network model used in FNR-Tree can represent only a single line segment, many entries and updates in the index structure occur. To overcome this disadvantage, Almeida et al. [31] proposed the MON-Tree where a hash structure pointing to the bottom level R-Trees was introduced. Generally, these tree-based indices are expensive for location updates because of the node reconstruction. Therefore, people usually prefer to use in-memory grid structure [15], [32], [33], [34] to achieve the efficiency when indexing the moving objects.

Darius et al. [13] systematically explored the performance of two basic moving objects indexes, i.e., the uniform grid and the R-tree, through comprehensive experimental comparison. According to their investigations, both grid and R-tree indexes tend to perform worse when the moving objects are non-uniformly distributed, which precisely justifies the necessity of our work, i.e., designing a new index method that can not only handle frequent updates but also support the efficient query processing. Later, Darius et al. [35] proposed an in-memory indexing method for moving objects which can exploit the parallelism offered by modern multi-core processors. However, they didn't consider the skewness of moving objects. In their recent work [36], they showed that the implementation details of the data structures and algorithms are very important for performance gain in the main-memory setting. Compared with their work, we focus on the high-level design and implementation for the techniques proposed in our work. Simba [37] is a distributed system built on Spark to support the scalable and efficient in-memory spatial query processing and analytics. Several classic index structures including R-trees embedded in Simba are optimized for Spark RDDs [38]. Simba can be viewed as a framework for general spatial queries while SIMkNN is proposed as a specified method to support the moving object kNN query.

7 Conclusion

This paper proposes SIMkNN, a scalable and in-memory method for *k*NN query processing over moving objects in road networks. SIMkNN is dual-index driven, where a hierarchical grid index is used to manage the moving objects and a R-tree is leveraged to store the connectivity information of the underlying road network. SIMkNN employs two consecutive phases, namely, *result space estimation* and *kNN retrieval*, to answer *k*NN queries. Compared with previous work, because of the nature of the hierarchical grid index, SIMkNN can adapt much better to the skewed distribution of moving objects. To derive the estimated area in the first step, SIMkNN proposes the cell communication technique that can keep each cell within the hierarchical grid being aware of its neighbours at any time. Additionally, a conservative space expansion algorithm has been proposed to slowly expand the search space. Extensive experimental evaluation shows that SIMkNN is more scalable than the baseline method for moving objects that are not uniformly distributed.

In our future work, we plan to i) further study the relations between different object distributions and the parameter settings, such as the split factor and cell capacity; ii) investigate other types of spatial queries and see whether the cell communication technique proposed in this paper could be used for improvement; and iii) extend our work to a distributed setting to answer the query in parallel, in order to support large scale *k*NN query processing.

References

[1] B. Cao, L. Alarabi, M. F. Mokbel, and A. Basalamah, "Sharek: A scalable dynamic ride sharing system," in *MDM*, 2015.

[2] S. Ma, Y. Zheng, and O. Wolfson, "Real-time city-scale taxi ridesharing," *IEEE Transactions on Knowledge and Data Engineering*, vol. 27, no. 7, pp. 1782–1795, 2015.

[3] "Didi kuaidi, ubers arch rival in china, confirms it raised $2b in fresh funding," http://techcrunch.com/2015/07/07/two-billionnnnnnnnn/, 2015.

[4] S. K. Begley, Z. He, and Y.-P. P. Chen, "Mcjoin: A memory-constrained join for column-store main-memory databases," in *SIGMOD*, 2012, pp. 121–132.

[5] R. Barber, G. Lohman, I. Pandis, V. Raman, R. Sidle, G. Attaluri, N. Chainani, S. Lightstone, and D. Sharpe, "Memory-efficient hash joins," in *VLDB*, 2014, pp. 353–364.

[6] H. Wang and R. Zimmermann, "Location-based query processing on moving objects in road networks," in *VLDB*, 2007, pp. 321–332.

[7] H. Wang and R. Zimmermann, "Snapshot location-based query processing on moving objects in road networks," in *SIGSPATIAL*, 2008, pp. 50:1–50:4.

[8] H. Wang and R. Zimmermann, "A novel dual-index design to efficiently support snapshot location-based query processing in mobile environments," *IEEE Transactions on Mobile Computing*, vol. 9, no. 9, pp. 1280–1292, 2010.

[9] A. Guttman, "R-trees: A dynamic index structure for spatial searching," in *SIGMOD*, 1984, pp. 47–57.
[10] K. C. Lee, M. Le, J. Harri, and M. Gerla, "Louvre: Landmark overlays for urban vehicular routing environments," in *VTC*, 2008, pp. 1–5.
[11] K. Zheng, Y. Zheng, X. Xie, and X. Zhou, "Reducing uncertainty of low-sampling-rate trajectories," in *ICDE*, 2012, pp. 1144–1155.
[12] R. Song, W. Sun, B. Zheng, and Y. Zheng, "Press: A novel framework of trajectory compression in road networks," *Proceedings of the VLDB Endowment*, vol. 7, no. 9, pp. 661–672, 2014.
[13] D. Šidlauskas, S. Šaltenis, C. W. Christiansen, J. M. Johansen, and D. Šaulys, "Trees or grids?: indexing moving objects in main memory," in *SIGSPATIAL*, 2009, pp. 236–245.
[14] H. Samet, "The quadtree and related hierarchical data structures," *ACM Computing Surveys*, vol. 16, no. 2, pp. 187–260, 1984.
[15] X. Yu, K. Q. Pu, and N. Koudas, "Monitoring k-nearest neighbor queries over moving objects," in *ICDE*, 2005, pp. 631–642.
[16] D. Papadias, J. Zhang, N. Mamoulis, and Y. Tao, "Query processing in spatial network databases," in *VLDB*, 2003, pp. 802–813.
[17] T. Abeywickrama, M. A. Cheema, and D. Taniar, "k-nearest neighbors on road networks: a journey in experimentation and in-memory implementation," *Proceedings of the VLDB Endowment*, vol. 9, no. 6, pp. 492–503, 2016.
[18] M. Mokbel, L. Alarabi, J. Bao, A. Eldawy, A. Magdy, M. Sarwat, E. Waytas, and S. Yackel, "A Demonstration of MNTG - A Web-based Road Network Traffic Generator," in *ICDE*, 2014, pp. 1246–1249.
[19] A. Guttman, *R-trees: a dynamic index structure for spatial searching*. ACM, 1984, vol. 14, no. 2.
[20] N. Roussopoulos, S. Kelley, and F. Vincent, "Nearest neighbor queries," in *ACM sigmod record*, vol. 24, no. 2, 1995, pp. 71–79.
[21] G. R. Hjaltason and H. Samet, "Distance browsing in spatial databases," *ACM Transactions on Database Systems*, vol. 24, no. 2, pp. 265–318, 1999.
[22] M. Sharifzadeh and C. Shahabi, "Vor-tree: R-trees with voronoi diagrams for efficient processing of spatial nearest neighbor queries," *VLDB*, vol. 3, no. 1, 2010.
[23] K. C. Lee, W.-C. Lee, B. Zheng, and Y. Tian, "Road: A new spatial object search framework for road networks," *IEEE Transactions on Knowledge and Data Engineering*, vol. 24, no. 3, pp. 547–560, 2012.
[24] R. Zhong, G. Li, K.-L. Tan, and L. Zhou, "G-tree: an efficient index for knn search on road networks," in *CIKM*, 2013, pp. 39–48.
[25] D. Delling and R. F. Werneck, "Customizable point-of-interest queries in road networks," *IEEE Transactions on Knowledge and Data Engineering*, vol. 27, no. 3, pp. 686–698, 2015.
[26] Z. Yu, Y. Liu, X. Yu, and K. Q. Pu, "Scalable distributed processing of k nearest neighbor queries over moving objects," *IEEE Transactions on Knowledge and Data Engineering*, vol. 27, no. 5, pp. 1383–1396, 2015.
[27] J. Chen and X. Meng, "Update-efficient indexing of moving objects in road networks," *Geoinformatica*, vol. 13, no. 4, pp. 397–424, 2009.
[28] K. Lee, R. K. Ganti, M. Srivatsa, and L. Liu, "Efficient spatial query processing for big data," in *SIGSPATIAL*, 2014, pp. 469–472.
[29] "Geohash," http://geohash.org/.
[30] E. Frentzos, "Indexing objects moving on fixed networks," in *SSTD*, 2003, pp. 289–305.
[31] V. T. De Almeida and R. H. Güting, "Indexing the trajectories of moving objects in networks," *GeoInformatica*, vol. 9, no. 1, pp. 33–60, 2005.
[32] H. D. Chon, D. Agrawal, and A. El Abbadi, "Range and k nn query processing for moving objects in grid model," *Mobile Networks and Applications*, vol. 8, no. 4, pp. 401–412, 2003.
[33] K. Mouratidis, D. Papadias, and M. Hadjieleftheriou, "Conceptual partitioning: an efficient method for continuous nearest neighbor monitoring," in *SIGMOD*, 2005, pp. 634–645.
[34] X. Xiong, M. F. Mokbel, and W. G. Aref, "Lugrid: Update-tolerant grid-based indexing for moving objects," in *MDM*, 2006, pp. 13–13.
[35] D. Šidlauskas, S. Šaltenis, and C. S. Jensen, "Parallel main-memory indexing for moving-object query and update workloads," in *SIGMOD*, 2012, pp. 37–48.
[36] D. Šidlauskas and C. S. Jensen, "Spatial joins in main memory: Implementation matters!" *Proceedings of the VLDB Endowment*, vol. 8, no. 1, pp. 97–100, 2014.
[37] D. Xie, F. Li, B. Yao, G. Li, L. Zhou, and M. Guo, "Simba: Efficient in-memory spatial analytics," in *SIGMOD*, 2016, pp. 1071–1085.
[38] "Apache spark project," http://spark.apache.org.

Bin Cao received his Ph.D. degree in computer science from Zhejiang University, China in 2013. He then worked as a research associate in Hongkong University of Science and Technology and Noah's Ark Lab, Huawei. He joined Zhejiang University of Technology, Hangzhou, China in 2014, and is now an Assistant Professor in the College of Computer Science. His research interests include spatio-temporal database and data mining.

Chenyu Hou received his BS in software engineering in Zhejiang University of Technology, Hangzhou, China, in 2016. He is now a postgraduate student of the Zhejiang University of Technology. His research interests are database and data mining.

Suifei Li received the MS in software engineering from Zhejiang University of Technology in 2016. Currently, she is a Java Development Engineer at Netease, Inc. Her research interest is spatial database.

Jing Fan received her B.S., M.S. and Ph.D. degree in Computer Science from Zhejiang University, China in 1990, 1993 and 2003. She is now a Professor of School of Computer Science and Technology at Zhejiang University of Technology, China. She is a Director of China Computer Federation (CCF), and Chairman of Chapter Hangzhou of CCF. Her current research interest includes middleware, virtual reality and visualization.

Jianwei Yin received his PhD degrees in computer science from Zhejiang University in 2001. He is currently a professor in the College of Computer Science at Zhejiang University. He is the visiting scholar of Georgia Institute of Technology, US, in 2008. His research interests include service computing and data management.

Baihua Zheng received the PhD degree in computer science from Hong Kong University of Science & Technology, China, in 2003. She is currently an associate professor in the School of Information Systems, Singapore Management University, Singapore. Her research interests include mobile/pervasive computing, and spatial databases.

Jie Bao is an associate researcher in Urban Computing Group at MSR Asia. He got his Ph.D. degree in Computer Science from University of Minnesota on 2014. He currently works on spatio-temporal data management on the cloud platform (i.e., Microsoft Azure). His main research interests include: Spatio-temporal Data Management/Mining, Database Systems, and Location-based Services.