Received August 3, 2019, accepted August 22, 2019, date of publication August 26, 2019, date of current version September 9, 2019.

Digital Object Identifier 10.1109/ACCESS.2019.2937667

Gridvoronoi: An Efficient Spatial Index for Nearest Neighbor Query Processing

CHONGSHENG ZHANG[1], GEORGE ALMPANIDIS[1], FAEGHEH HASIBI[2], AND GAOJUAN FAN[1]
[1]School of Computer and Information Engineering, Henan University, Kaifeng 475001, China
[2]Institute of Computing and Information Sciences, Radboud University, EC 6525 Nijmegen, The Netherlands

Corresponding author: Gaojuan Fan (fangaojuan@126.com)

This work was supported in part by the European Research Consortium for Informatics and Mathematics (ERCIM), and National Natural Science Foundation of China (NSFC) under Grant No. 41401466.

ABSTRACT In this paper, based upon Voronoi Diagram, we propose *GridVoronoi* which is a novel spatial index that enables users to find the spatial nearest neighbour (NN) from two-dimensional (2D) datasets in almost O(1) time. *GridVoronoi* augments the Voronoi Diagram with a virtual grid to promptly find out (in a geometric space) which Voronoi cell contains the query point. It consists of an off-line data pre-processing phase and an on-line query processing phase. In the off-line phase, the digital geographical space is partitioned with a Voronoi Diagram and a virtual grid, respectively. Next, for each square unit (i.e., grid cell), the corresponding Voronoi cells that contain or intersect with this square are derived and kept in a hashmap-like structure. In the on-line phase, for each real-time spatial NN query, the algorithm first identifies which virtual square(s) contain(s) this query; then looks up the hashmap structure to find the corresponding Voronoi cell(s) for this grid cell and the final result for the query. Overall, *GridVoronoi* significantly reduces the time complexity in finding spatial NN in 2D space, thus improves the efficiency of real-time spatial NN queries and Location Based Services.

INDEX TERMS Geospatial analysis, nearest neighbour methods, query processing, spatial databases.

I. INTRODUCTION

Due to the massive spread of smartphones and the development of precise positioning techniques, location-based services have become increasingly popular in everyday life. More and more Web resources are being Geo-tagged; the number of Geo-referenced Web objects, such as restaurants and hotels associated with location information and textual descriptions, has been growing exponentially. In recent years, a significant amount of research work has focused on managing and mining such Geo-referenced Web objects, in which one challenging task is the processing of database queries that take location into account (often referred to as spatial queries, or location-aware queries).

Spatial (K) **N**earest **N**eighbor (NN) search techniques can be used in many different real-life applications, including (but not limited to) Location-Based Services (LBS), urban space planning [1], route planning [2], [3], positioning based on Internet of Things (IOT) [4]–[6], location verification/privacy protection [7]–[10], spatial data mining [11]–[15], etc.

The associate editor coordinating the review of this article and approving it for publication was Waleed Alsabhan.

In this paper, we investigate how to improve the efficiency of spatial NN query processing, which is a fundamental issue for top-k spatial keyword queries [16] and preference queries [17], location-aware recommendation [18], etc. The Geo-location of a query is often modelled as a point located in a geographical area where all the data points reside. In this paper, we focus on geographical areas that consist of two-dimensional (2D) data sets that are widely available, and frequently queried in Geo-spatial applications and Location Based Services.

While a significant amount of research work on location-aware query processing has focused on the optimisation aspects of finding top-k results [16], [17], [19], very little attention has been paid on how to improve the processing efficiency of a single NN query. NN querying is often assumed to be an issue that has been thoroughly studied. When searching for location-aware nearest neighbours, almost all existing work relies on R-Tree-like structures to access the local neighbourhood of a spatial NN query. However, using R-Tree-based structures, the time complexity for arriving to an NN query's local neighbourhood in a geometric space is O(log n) (n is the total number of objects/data points), which is very expensive. Moreover, all nodes/entries intersecting this neighbourhood must be checked to obtain the final NN object, leading to the unnecessary examination of many unrelated entries. Thus, optimizing spatial NN querying is a critical research issue that deserves more attention.

As mentioned above, existing approaches need O($\log n$) time to access an NN query's corresponding neighbourhood in the geographical area, which is highly time-consuming. The ideal situation is to arrive to query's matching location in the geographical area and find the nearest spatial object in O(1) time. If there is an algorithm which can turn this expectation into reality, then the efficiency for real-time spatial NN query processing can be substantially improved.

To this end, we propose a method which is able to find an NN query's nearest spatial object in 2D datasets in almost O(1) time. We refer to this method as *GridVoronoi* since it uses a Voronoi Diagram [20], [21] blended with a virtual square grid for quickly accessing the corresponding location of the query in the geographical area and finding the nearest spatial object in almost O(1) time. Voronoi Diagram structure is highly efficient in exploring the local neighbourhood in a geometric space, but it lacks an efficient access method that can help it locate this neighbourhood promptly. For this reason, we augment Voronoi Diagram with a virtual grid. This grid is virtual because we do not need to establish a physical grid in the geographical area. There are generally two phases in *GridVoronoi*: the offline pre-processing phase and the online query processing phase.

In the offline pre-processing phase, we first partition the digital geographical area using Voronoi Diagram, and each Voronoi cell contains only one spatial data point. Next, with a virtual grid, we virtually divide the same geographical area into square units. With the virtual grid, we can immediately calculate the corresponding virtual square (in the virtual grid) that covers an NN query's spatial location. We also propose an efficient algorithm that pre-computes for each virtual square which Voronoi cell(s) contain(s)/intersect(s) it and keeps the correspondence in a hashmap.

In the online query processing phase, for each spatial NN query, *GridVoronoi* first calculates which square unit of the virtual grid contains this query. It next finds in the above hashmap the Voronoi cell that contain(s)/intersect(s) this square unit and returns the corresponding spatial data point for this Voronoi cell, as the query result.

In Figure 1, we demonstrate the main idea of *GridVoronoi*. It uses Voronoi Diagram and a virtual grid of square units to divide the 2D geographical area. $v_0 \ldots v_{19}$ are *20* Voronoi cells generated by Voronoi Diagram, while the square units (with light gray color) are part of the virtual grid. Q_1 and Q_2 are two real-time spatial NN queries. Q_1 is inside v_9, whereas Q_2 is at the intersection of v_8, v_9 and v_{13}. We observe that a Voronoi cell may contain many virtual squares and it can intersect with some virtual squares as well. For Q_1, by simple calculations in the virtual grid, we can immediately find the corresponding virtual square it locates. If *GridVoronoi* is able to locate the Voronoi region (v_9) corresponding to the virtual square, then the data point in v_9 must be the answer

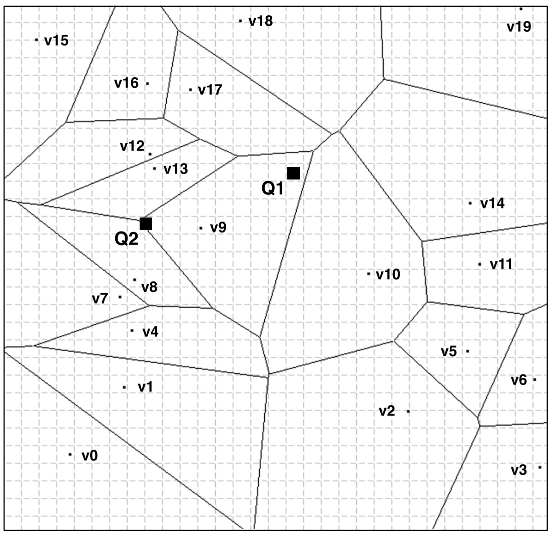

FIGURE 1. A Toy Example for *GridVoronoi*.

for Q_1. For Q_2, we also calculate the virtual square that covers the query location. If *GridVoronoi* knows the square is at the intersection of v_8, v_9 and v_{13}, then it only needs to check the data points in these three Voronoi cells to find out the final nearest neighbour for Q_2. For both queries, *GridVoronoi* will be extremely efficient because it only needs to check very few Voronoi cells. An important problem that *GridVoronoi* needs to investigate is how to compute which virtual squares are contained by the Voronoi cells or which virtual squares intersect the Voronoi cells, which we will address in the following sections.

The rest of the paper is organised as follows. Section II reviews the prominent spatial NN search approaches. In Section III, we introduce the *GridVoronoi* approach which enhances the Voronoi diagram with a virtual grid to immediately locate the Voronoi cell that contains a query point. In Section IV, we address the issue of optimizing the side-length of the grid cells for fast NN query processing. Then in Section V, we discuss the complexity of *GridVoronoi*. In Section VI, we extend *GridVoronoi* to process K nearest neighbour queries. In Section VII, we experimentally compare *GridVoronoi* with two well-established approaches on real-world datasets. Finally, Section VIII concludes the paper.

II. RELATED WORK

The spatial nearest neighbour search problem has been extensively studied in the literature. Existing algorithms can be grouped into four categories: R-Tree based, Voronoi Diagram based, Grid-partition based and Hybrid approaches.

A. R-TREE BASED APPROACHES

For R-Tree based approaches for (K)NN query processing, *Best-First Search* (BFS) [22] has been the prevalent technique for processing nearest neighbour queries [23]. BFS employs a priority queue to store all nodes that need to be explored through the search process. It traverses from the root of the R-Tree down to the leaves. For each minimum bounding rectangle (MBR) it has visited, it computes the bounded distances between the query point and the MBR, then inserts the entry of each visited MBR into a heap and follows the one closest to the query point.

In recent years, a significant amount of work [19], [24]–[26] has been focused on processing spatial keyword queries, that is, queries that specify both a location and a set of keywords. *IR*^{2}*-tree* [24] is a well-known approach that efficiently answers top-k spatial keyword and preference queries. It consists of an R-Tree and a signature tree to facilitate both top-k spatial queries and top-k spatial keyword queries. Similarly, *IR-tree* [25] is an R-Tree extended with inverted files for processing the location-aware top-k text retrieval queries. Each leaf node in the R-Tree is associated with an inverted file with the text descriptions of the objects stored in this region. Essentially, R-Tree is leveraged for spatial proximity querying, whereas the inverted file is used for text retrieval.

B. VORONOI DIAGRAM BASED APPROACHES

Besides R-Tree, many state-of-the-art approaches utilise Voronoi Diagram to enhance KNN query processing because Voronoi diagrams are efficient data structures for exploring a local neighbourhood in a geometric space.

The most related approach to *GridVoronoi* is the *VoR-Tree* structure proposed in [27]. Both methods utilise Voronoi diagram to make full use of its power in exploring a local neighbourhood in a geometric space. Moreover, both *GridVoronoi* and *VoR-Tree* can help Voronoi Diagram come to the neighbourhood of a query quickly. However, there are two main differences between them. First of all, *VoR-Tree* utilises an R-Tree to access to the query's neighbourhood in the geographical space, but the time complexity for R-Tree to find the neighbourhood is $O(\log n)$. In contrast, *GridVoronoi* uses only (almost) $O(1)$ time to reach this neighbourhood because it simply calculates the corresponding virtual square for NN query points. Second, to speed up the computation for NN, *VoR-Tree* associates each data point in the R-Tree with a structure that contains the Voronoi cell for this data point and its neighbouring Voronoi cells. Since the neighbourhood derived from the first step may contain several overlapped MBRs in the R-Tree, there may be many data points in this neighbourhood that *VoR-Tree* needs to check for each data point whether the Voronoi cell in the associated structure contains the query point. With *GridVoronoi*, we only need to check in the hashmap for the corresponding Voronoi cell of the square calculated in the first step.

V^*-Diagram [28] is a well-known work that utilises Voronoi Diagram for processing moving KNN queries. It computes a safe region by jointly considering the query location, the data objects and the current search space. By doing so, the computation cost for providing continuous answers to the moving KNN queries can be greatly reduced. *GridVoronoi* can also be applied to processing moving NN queries. There are two cases to consider. The first is that the virtual square for the current query is contained by a Voronoi cell. In this case, they check whether the virtual square for the new query is still being covered by the Voronoi cell. If yes, there is no change in the query answer. Otherwise, it will be processed as the second case. The second case is that the virtual square for the current query is at the intersection of two or more Voronoi cells. In this case, they only compute the distances between the query's new location and these Voronoi cells. Hence, with *GridVoronoi* the overhead for maintaining continuous answers for moving NN queries will be very low.

The authors of [29] also employ Voronoi diagram for processing moving KNN queries. In their method, instead of continuously checking the validity of the safe regions and recomputing them if invalidated, they use a small set of safe guarding objects (influential set). This enables users to avoid the high pre-computation cost of order-k Voronoi diagram by computing it locally and on-the-fly.

C. GRID-PARTITION BASED APPROACHES

Another work related to ours is the *conceptual partitioning* method (CPM) proposed in [30] for monitoring continuous NN queries. In CPM, the data objects are indexed by a main-memory grid with square units and each square is associated with the list of objects residing therein. With the help of the square grid, in continuous NN monitoring people only need to consider the minimal set of square units to retrieve the NN when the query moves. Both *GridVoronoi* and CPM use grids to partition the space, but the difference is that the former builds a correspondence between each virtual square and the Voronoi cell(s) that contain(s) this virtual square. This way, when an NN query comes, we just need to calculate the matching virtual square that contains the NN query, then search in the hashmap and return the data point in the retrieved Voronoi cell. Using CPM, for each NN query, people have to check many local squares in order to make sure that they can get the true nearest neighbour, which consumes more time than *GridVoronoi*.

A work that discusses scalable NN query processing is [31], where a novel distributed spatial data index (Inverted Grid Index), which is a combination of inverted index and grid partition, is constructed. The authors show that the index constructing time is over 25% less than R-tree and Voronoi-based index. The authors study the influence on efficiency of KNN query when the width of the grid cell varies. They deduce that the optimal value of cell width depends on the distribution of data points in specific dataset.

D. HYBRID APPROACHES

Another category of approaches for (K)NN query processing utilise grid-partition indices combined with Voronoi Diagram based approaches which can be pre-computed offline to improve the efficiency of NN query processing [32].

In [33], the authors propose the "grid-partition" index which is essentially a combination between one-row (stripe-like) grid cells and Voronoi Diagram, as it associates objects with grid cells for fast query processing. The differences between [33] and our work mainly lie in two aspects. First, depending on the dataset size and shape, in [33], a grid cell can intersect with dozens of voronoi cells, whereas in our proposed method, the majority of the grid cells (squares) intersect (or contain/being contained by) 1 or 2 Voronoi cells. The number of associated objects (data points) greatly influences the NN query processing efficiency. Second, for our work, we prove that, the NN query processing efficiency is monotonically decreasing with the side-length of the grid cells (squares), which can not be guaranteed in [33].

Reference [34] is another hybrid method related to ours. Their proposed method, which provides approximate answers for range NN queries, aims to balance the performance trade-off between query response time and the quality of answers for mobile users in mobile cellular network, through a user-specified "approximation tolerance level" parameter. Their solution is based on Voronoi diagram and an incomplete pyramid structure as the access method to index the Voronoi cells. There are several differences between [34] and our work. First of all, the former aims to process range NN queries, while our work investigates how to speed up NN query processing. Second, [34] can not completely guarantee the correctness of the query result, when computing the intersections between a grid cell in the current level k and those in the lower (i.e., $k + 1$) level for range NN queries. In contrast, our work can provide exact answers for NN queries. Furthermore, our work gives techniques on deriving the intersecting and containing relationships between the grid cells and the Voronoi cells. More importantly, our work provides theorems on the monotonically decreasing relationship between the side-length of the grid cells and the query processing efficiency.

In summary, most of the existing work relies on R-Tree based structures to reach the neighbourhood of a spatial query's corresponding location in geographical space where all the data points reside in, leading to a time complexity of O(log n).

III. GRIDVORONOI APPROACH

In this section, we introduce the *GridVoronoi* approach for efficient nearest neighbour search in a two-dimensional (2D) datasets. There are two general phases in *GridVoronoi*. The first is an offline pre-processing phase in which we build the structures needed by *GridVoronoi*, while the second phase is mainly focused on online NN query processing. Together, there are five specific steps which we will elaborate in the following.

A. OVERALL FRAMEWORK

In this subsection, we show how we use *GridVoronoi* to index the 2D spatial data and process real-time spatial NN queries. There are two phases in *GridVoronoi* which are the offline data pre-processing phase and the online query processing phase. In the offline phase, we organise the spatial data and offer a fast path for a query to find the NN object to its location

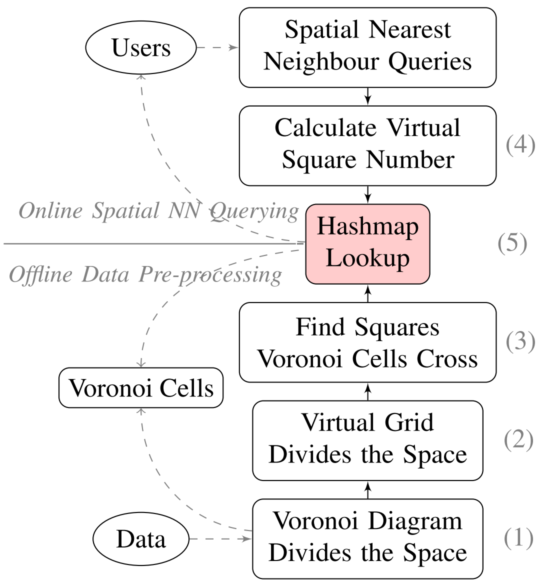
Users
Spatial Nearest Neighbour Queries
Calculate Virtual Square Number
(4)
Online Spatial NN Querying
Hashmap Lookup
(5)
Offline Data Pre-processing
Find Squares Voronoi Cells Cross
(3)
Voronoi Cells
Virtual Grid Divides the Space
(2)
Data
Voronoi Diagram Divides the Space
(1)

FIGURE 2. Overall framework.

in the 2D space. In the online query processing phase, for each spatial NN query we first compute its location in the digital geographical area, then through the stored correspondences between a location and the nearest spatial object made possible by the offline phase, we directly return the nearest spatial data point to this location as the query answer.

In Figure 2, steps (1), (2) and (3) happen in the offline pre-processing phase. In step (1), we use the Voronoi Diagram to partition the 2D geographical area, which will be detailed in subsection III-B. Then in step (2), we virtually partition the same geographical area using a grid, the details of which will be described in subsection III-C, whereas the issue on how to set an optimised side-length for the grid cells is illustrated in section IV. To compute the correspondences between a location and the nearest spatial object, in step (3) we design a method that determines the virtual square units (also called "virtual squares" or "grid cells" hereafter) that a given Voronoi cell contains/crosses. We provide details of this method in subsection III-E. Steps (4) and (5) are for online query processing, in step (4) we calculate the virtual square unit that covers the query point (given in subsection III-D), while in step (5) we look up the hashmap and retrieve the Voronoi cells corresponding to the virtual square unit. From the data points in these Voronoi cells, we select the one nearest to the query point as the query result.

B. USING VORONOI DIAGRAM TO PARTITION THE SPACE

Given a set of data points in the space, a general Voronoi Diagram [21] partitions the space into disjoint regions. Each disjoint region, referred to as a Voronoi cell, is a convex polygon. Each Voronoi cell may have a number of surrounding Voronoi cells, each of which shares at least one side/edge with this Voronoi cell. By definition of the Voronoi diagram, the center of each Voronoi cell is resided by a data point (POI) from the dataset. That is, each Voronoi cell must contain a data point (POI) which is centered in the Voronoi cell. Moreover, according to the properties of Voronoi diagram (which can be found in the references [21]), the nearest neighbour for each query point located inside the area of the Voronoi cell must be the corresponding data point (POI) which centers in the Voronoi cell. If a query point is on the side of two neighbouring Voronoi cells, then the distances from the query point to the two data points in the two Voronoi cells are the same.

Once we know the query point q is in which Voronoi cell, i.e., which Voronoi cell covers this query point, then we immediately know the data point of this Voronoi cell is the nearest neighbour of q. Even when a query point is on the side of a Voronoi cell, we only need to return the data points of the two Voronoi cells sharing this side [21]. Overall, once the Voronoi cell that contains the query point is known, Voronoi diagram can simultaneously remove the need for re-scanning all the data points in the space and guarantee that we can promptly find the NN object to the query point.

But how do we know which Voronoi cell contains the query point? If all the Voronoi cells are equal-sized squares or rectangles, then through dividing the coordinate position by the side length of the square or rectangle, we can immediately calculate the query locates in which square or rectangle cell. Unfortunately, Voronoi cells are usually irregular convex polygons, their sizes vary as well. Hence, we need a method that can quickly determine whether a query point is contained in which Voronoi cell. A straightforward solution is to traverse all the Voronoi cells until we find one that contains the query point. Another solution is the *VoR-Tree* proposed in [27] which utilises an R-Tree structure to find the query's corresponding neighbourhood in the space. The straightforward approach is obviously very time-consuming. *VoR-Tree* is more efficient than the straightforward approach, but as already analysed in Section II, it is still costly. Thus we propose *GridVoronoi* in which we add a virtual grid to Voronoi Diagram to quickly compute the Voronoi cell that contains the query point. Prior to using the virtual grid, we need to set an appropriate side-length for the grid cells in the virtual grid, which we will address in Section IV.

C. NUMBERING THE SQUARES OF THE VIRTUAL GRID

Given the side-length of the grid cells, denoted as l, we address how to virtually partition the 2D space using a virtual grid, and how to number the virtual square units. Let the 2D plane be a bounded rectangular area $\mathbb{R}$, the matter now is how to partition $\mathbb{R}$ with a virtual grid of squares. Let the lower left vertex of $\mathbb{R}$ be $P_0(x_0, y_0)$, the upper left, upper right and lower right vertices be $P_1(x_0, y_1)$, $P_2(x_1, y_1)$ and $P_3(x_1, y_0)$, respectively. Let *ceil(x)* be a function that rounds the element x to the nearest integer towards infinity, and $floor(x)$ be a function that rounds the element x to the nearest integer towards minus infinity.

We divide $\mathbb{R}$ into m-by-n virtual square units: for each row there are $m = ceil(\frac{x_1-x_0}{l})$ virtual squares; while for each column, there are $n = ceil(\frac{y_1-y_0}{l})$ virtual square units. If $\mathbb{R}$ and the virtual square units are regarded as a matrix, then $\mathbb{R}$ is converted into a m-by-n matrix $\mathbb{M}$. Accordingly, the virtual square numbering problem is equivalent to the problem of numbering the elements in $\mathbb{M}$.

For convenience of explanation, we hereafter assume the coordinates of the lower left point of $\mathbb{R}$ be (0,0), that is, $x_0 = 0$ and $y_0 = 0$. We number the virtual square units in bottom-to-up and left-to-right order. Let element $\mathbb{M}(x, y)$ represent the element at the *xth* row and *yth* column of $\mathbb{M}$, then the number for the corresponding virtual square is $m \times x + y$. For instance, in Figure 3, $m = n = 8$, element $\mathbb{M}(0, 0)$ represents the *0th* element of the *0th* row in $\mathbb{M}$, then the number for this element and the corresponding virtual square is 0; similarly, element $\mathbb{M}(2, 3)$ represents the *3rd* element of the *2nd* row, so the number for this element and the corresponding virtual square unit is $8 \times 2 + 3 = 19$.

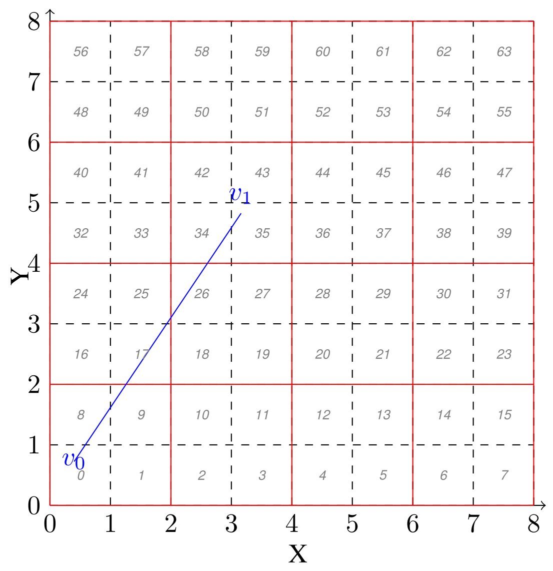

FIGURE 3. Number the squares and find the squares that the edge $v_0 \rightarrow v_1$ intersects.

It should be noted that the rightmost and uppermost virtual squares will be numbered in the same way, even though their sizes may be smaller than l-by-l.

D. FINDING THE VIRTUAL SQUARE FOR A QUERY

After the two-dimensional space is divided into virtual square units, we now study how to find the corresponding virtual square for a user-specified NN query *Q(x,y)*. We present our method in Algorithm 1.

We use Formula 1 to compute the number (i.e., the identifier) of the matching virtual square/grid cell for query *Q(x,y)*, if *Q(x,y)* is not on a side of the virtual square. Lines 2-4 of Algorithm 1 are the pseudo-code in accordance with Formula 1.

$$number = m \times (ceil(\frac{y-y_0}{l}) - 1) + (ceil(\frac{x-x_0}{l}) - 1) \qquad (1)$$

Algorithm 1: FindVirtualSquare: Find the Virtual Square(s) that Contain(s) an Input Query Point

```
Input: the coordinates of a query point $p$, $x$ and $y$;
       square side length $l$
Output: the identifier(s) of the virtual square(s)
        that contain(s) $p$
1  begin
2      $u \leftarrow l \times (ceil(x/l) - 1)$;
3      $b \leftarrow l \times (ceil(y/l) - 1)$;
4      $sq[0] \leftarrow m \times b + u$;
5      if $x = u$ then
6          if $y = b$ then
7              $sq[1] \leftarrow m \times (b+1) + u$;
8              $sq[2] \leftarrow m \times b + u + 1$;
9              $sq[3] \leftarrow m \times (b+1) + u + 1$;
10         else
11             $sq[1] \leftarrow m \times b + u + 1$;
12     else
13         if $y = b$ then
14             $sq[1] \leftarrow m \times (b+1) + u$;
15     return sq;
16 end
```

If *Q(x,y)* is on a side of the virtual square, or it coincides with a vertex of the square, then we consider the virtual square units sharing the side or vertex.

Overall, we only need to check which Voronoi cell(s) contain(s) the corresponding virtual square. Since such correspondences are stored in a hashmap, we can promptly find the nearest data point to the spatial NN query.

E. FINDING THE VIRTUAL SQUARES A VORONOI CELL CROSSES/CONTAINS

Determining the Voronoi cells a virtual square intersects is an important task in *GridVoronoi*. Given a Voronoi cell, we need a method to find out the virtual squares intersecting with it.

A Voronoi cell consists of several vertices and edges (i.e., sides), so we first need to study the problem of finding the grid cells that a given edge crosses. Let the vertices of a Voronoi cell be orderly stored in accordance with the adjacency relationship. In Figure 3, let $V_0(x_0, y_0)$, $V_1(x_1, y_1)$ be the two endpoints of a Voronoi edge E, the goal is to return the numbers (i.e., the identifiers) of the virtual squares that E crosses.

We introduce an efficient method to find out the virtual squares that E crosses. The general idea is to first calculate the intersections of E and the squares, next infer from the intersections the specific virtual squares that E crosses.

Starting from one endpoint of E, we successively compute the intersections of E and vertical lines whose x coordinate values are the integer times of l and range from x_0 to x_1. Similarly, we calculate the intersections of E and horizontal lines whose y coordinate values are the integer times of l and range from y_0 to y_1. We also consider the cases when E is vertical or horizontal. After we insert all the intersections and the two endpoints into a set *cp*, we sort data points in *cp* by their y coordinate values. We next compute the midpoints of two adjacent data points in *cp* and for each midpoint find the corresponding virtual square that contains it. Finally, we return the numbers of these squares.

The reason that we calculate the midpoints is that, if a line intersects a virtual square, there will be two intersections and the midpoint of them is contained by only one virtual square. This way, each midpoint corresponds to only one virtual square that E crosses. To get all the virtual squares that E crosses, we just need to find out the squares that contain the midpoints.

After we calculate all the squares that the edges of the Voronoi cell crosses, the Voronoi cell should be wrapped by these surrounding virtual squares. Then the virtual squares inside the fenced area must include all the candidate squares that the Voronoi cell contains.

Algorithm 2: HashSquares: Store the Correspondences between the Virtual Square(s) and a Voronoi Cell in a Hashmap

```
Input: the vertex set VS ofa Voronoi cell VC
Output: a hashmap that stores the correspondences
        between the virtual square(s) and VC
1  begin
2      $vid \leftarrow$ the identifier of $VC$;
3      $v_0 \leftarrow VS[0]$;
4      for $i \leftarrow 1$ to $VS.size() - 1$ do
5          $v_1 \leftarrow VS[i]$;
6          $sq \leftarrow FindSquaresCrossed(v_0, v_1)$;
7          insert all elements of $sq$ into $IS$;
8          $v_0 \leftarrow v_1$;
9      $sq \leftarrow FindSquaresCrossed(VS[i], VS[0])$;
10     insert all elements of $sq$ into $IS$;
11     $IS \leftarrow sort(IS)$;
12     $f_0 \leftarrow floor(IS[0]/n)$;
13     insert $(f_0, vid)$ into hashmap;
14     for $j \leftarrow 1$ to $IS.size() - 1$ do
15         $f_1 \leftarrow floor(IS[i]/n)$;
16         insert $(f_1, vid)$ into hashmap;
17         if $f_0 = f_1$ then
18             if $IS[i] - IS[i-1] > 1$ then
19                 for $c \leftarrow IS[i] + 1$ to $IS[i-1] - 1$ do
20                     insert $(c, vid)$ into hashmap;
21         else
22             $f_0 \leftarrow f_1$;
23 end
```

Algorithm 2 addresses the problem of finding all the virtual squares that a Voronoi cell contains or intersects. We first find all the virtual squares intersecting with the edges of a

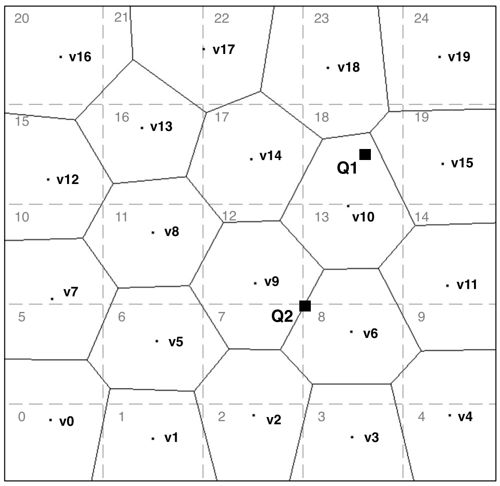

FIGURE 4. ***GridVoronoi*** **on 2D data points.**

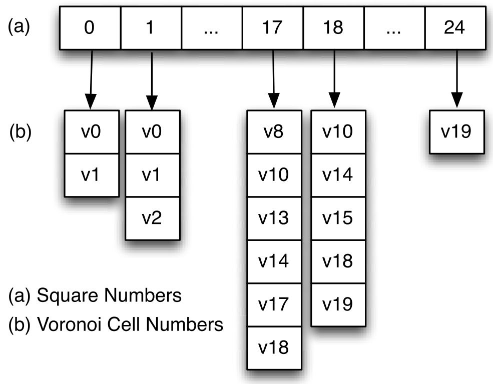

FIGURE 5. The hashmap that maintains the correspondences between the grid cells and the voronoi cells.

Voronoi cell; next, we sort the virtual squares by their identifiers, then find the virtual squares the Voronoi cell contains. The main observation is that, **virtual squares between two surrounding squares (that intersect the Voronoi cell) on the same row (i.e. along the same horizontal line) must be contained by the Voronoi cell.** So we iteratively calculate the surrounding virtual squares that are on the same row, then store in a hashmap the correspondences between the Voronoi cell and the virtual squares that are between two surrounding virtual squares, i.e., the numbers of these virtual squares are between that of the two surrounding virtual squares on the same row. Such a strategy makes Algorithm 2 very efficient in finding the virtual squares contained in a Voronoi cell.

In short, Algorithm 2 helps us promptly identify the virtual squares that a Voronoi cell contains. We keep a hashmap that stores the correspondences between the Voronoi cell and the intersecting virtual squares, as well as the virtual squares contained in the Voronoi cell. With such a hashmap, we can rapidly find the matching Voronoi cell for the virtual square that contains the NN query.

Examples 1 and 2 are two cases that show separately the offline data pre-processing phase (i.e., the offline index creation phase) and the online NN query processing phase in *GridVoronoi*.

Example 1: In Figure 4, we use GridVoronoi to split the 2D data. We also number the virtual squares, ranging from 0 to 24. In Figure 5, we show the hashmap structure that keeps the correspondences between the virtual squares and the Voronoi cells that contain/intersect the squares. For a virtual square that intersects two or more Voronoi cells, e.g., square 18 intersects v_{10}, v_{14}, v_{15}, v_{18} and v_{19}, the value set in the hashmap for the virtual square will include the numbers of all these Voronoi cells. But if a virtual square is contained in a Voronoi cell, e.g., square 24 is contained in v_{19}, then in the hashmap, there will be only one Voronoi number in the value set for the square. The final hashmap will be used in the online query processing phase, as shown in Example 2.

Example 2: Assume query points Q_1 and Q_2 in Figure 4 are two real-time NN queries. In Figure 5, we have the hashmap structure, generated in the offline data pre-processing phase as shown in Example 1, that keeps the correspondences between the virtual squares and the Voronoi cells that contain/intersect the squares. For Q_1, we first calculate the number of the grid cell that it resides using Algorithm 1, which is 18. Then we lookup 18 in the hashmap in Figure 5 and find the Voronoi cells v_{10}, v_{14}, v_{15}, v_{18} and v_{19}. We next retrieve the data points in these 5 Voronoi cells and compute the nearest data point to Q_1. Similarly, we can find the nearest data point(s) to Q_2. Notice that since Q_2 happens to be the common vertex of squares 7, 8, 12 and 13, we need to look up all these square numbers in the hashmap, as illustrated in subsection III-D. The final query answers for Q_1 and Q_2 will be the data point associated with v_{10}, and the two data points contained in v_6 and v_9, respectively.

We note that, in our implementation we employ the C++ Standard Template Library for the hashmap structure. Each key of the hashmap represents the identifier of a virtual square, while the value corresponding to this key is the identifier of the Voronoi cells that contains/intersect with the square. If there are 2 or more such voronoi cells, we use a linked list structure to connect them and stored as the value for the key, as shown in Figure 5.

IV. DETERMINING THE SIDE-LENGTH OF THE GRID CELLS

Now we study the influence of the slide length (grid cell length) l to the performance of *GridVoronoi* for processing NN queries and whether there is an optimal value for l to achieve the best performance.

The side-length of the grid cells is a very important parameter. If the side-length is too large, then each grid cell will include or intersect with so many Voronoi cells that it will be computation-demanding to check all of them for each query. On the other hand, if the side-length is too small, then there will be such a great amount of virtual squares (grid cells) that storing the correspondences between the grid cells and the Voronoi cells in a hashmap will consume a lot of memory space.

To find the NN of a query point q, for the grid cell in which q is located, we need to check the corresponding Voronoi cells of this grid cell (details given in subsection III-E). There are three situations to be considered, as described in Lemma 1.

Lemma 1: To find the NN of a query point q through GridVoronoi: (1) for no more than $2n-5$ grid cells (let $\mathbb{T}$ be the union of such grid cells and t be the total number, $t \leq 2n-5$), we have to check three or more Voronoi cells; (2) for the grid cells that intersect the Voronoi cells (let $\mathbb{W}$ be the union of such grid cells and w be the total number), we only need to check two Voronoi cells; (3) for the grid cells that are contained in the Voronoi cells (let $\mathbb{U}$ be the union of such grid cells and u be the total number), we only need to check one Voronoi cell.

Let y be the number of Voronoi cells that we need to check for an NN query q, x be the corresponding grid cell for q. Then,

$$y = \begin{cases} 1 & x \in \mathbb{U} \\ 2 & x \in \mathbb{W} \\ > 2 & x \in \mathbb{T} \end{cases} \tag{2}$$

There are at most $2n-5$ vertices in the Voronoi diagram [21]. Thus, if q is located in a grid cell that contains a vertex, then we should check all the Voronoi cells sharing this vertex.

For a given dataset, the number of Voronoi vertices is fixed, then $\mathbb{T}$ and t are also fixed; the size of the bounded rectangular area $\mathbb{R}$ for the dataset (see subsection III-C), denoted as r, is also fixed. Then,

$$t + w + u = \frac{r}{l^2} \tag{3}$$

To improve the efficiency for processing q, we expect the probability that a grid cell corresponds to only one Voronoi cell to be as high as possible, i.e., the proportion of $\mathbb{U}$ in $\mathbb{R}$ to be as large as possible.

Let $P(q \in \mathbb{U})$ represent the probability that a query point q locates in a grid cell that corresponds to only one Voronoi cell; X_i be the number of squares a side intersects; b be the total number of sides. With Equation 3, we infer that

$$\begin{aligned} P(q \in \mathbb{U}) &= 1 - \frac{w+t}{\frac{r}{l^2}} \\ &= 1 - \frac{l^2 t + \sum_{i=1}^{b} l^2 X_i}{r} \end{aligned} \tag{4}$$

In Equation 4, $l^2 t$ represents the combined area for the grid cells that belong to $\mathbb{T}$, whereas $\sum_{i=1}^{b} l^2 X_i$ is the total area of the grid cells in $\mathbb{W}$. For a given dataset, r, b and t are all fixed. It is clear that, to have a maximum $P(q \in \mathbb{U})$, we only need to make l as small as possible.

Let y be the number of Voronoi cells that we need to check for a query q. For $q \in U$, where a query q corresponds to only one Voronoi cell, the cost of computations is low. The same goes for query $q \in W$, where the query corresponds to two Voronoi cells. Hence, we only need to minimise the number of query points corresponds to more than two Voronoi cells. To this goal, let $P(q \in T)$ be the probability of the query q locating in a grid cell, where the grid cell contains a Voronoi vertex. Let l be the side length of the grid cells and r be the size of bounded rectangular area for the 2D dataset, then: $P(q \in T) = \frac{(2n-5)*l^2}{r}$. The numerator of this equation denotes the total area of grid cells belong to T. For a given dataset, $(2n-5)$ and r are fixed. Thus, to have a minimum $P(q \in T)$, we simply need to put l as small as possible.

Therefore, there is no single optimal value for l ($l > 0$) to achieve the best $P(q \in \mathbb{U})$. **The efficiency of *GridVoronoi* in Nearest Neighbour Query Processing is monotonic with respect to the side-length l: the smaller the value of l, the more efficient *GridVoronoi* will be in NN processing.**

In the following, we address two alternatives to derive an appropriate side-length for GridVoronoi. However, before we give details on these two approaches, we note that the values to be derived by these approaches will not be optimal, because a smaller side-length l is always preferred, given the fact that *GridVoronoi* is monotonic with respect to the side-length l. Nevertheless, these two methods will help users promptly pick an appropriate value for l (the upper bound or lower bound of value of l), given user-specified constraints or hardware constraints.

A. PARAMETER-DRIVEN SIDE-LENGTH FOR THE GRID

$P(q \in \mathbb{T})$ represents the probability that a query point q locates in a grid cell that corresponds to three or more Voronoi cells. Then we have:

$$P(q \in \mathbb{T}) \approx \frac{t * l^2}{r} \tag{5}$$

Since $t \leq 2n-5$, we have

$$P(q \in \mathbb{T}) \leq \frac{(2n-5) * l^2}{r} \tag{6}$$

For a given dataset, n, t and r are fixed, then l is the only variable. Thus, the smaller the value of l, the lower the value of $P(q \in \mathbb{T})$. For instance,

- If $l = \frac{1}{\sqrt{n}}$, then $P(q \in \mathbb{T}) = \frac{t}{r*n} < \frac{2}{r}$;
- If $l = \frac{1}{n}$, then $P(q \in \mathbb{T}) = \frac{t}{r*n^2} < \frac{2}{r*n}$.

Let Γ be a user-specified variable that defines the maximum value of $P(q \in \mathbb{T})$, i.e., the maximum percentage of grid cells that intersect three or more Voronoi cells. Then we have:

$$\Gamma \geq P(q \in \mathbb{T}) \tag{7}$$

Based on Formula 5 and 7, we infer Formula 8:

$$l \leq \sqrt{\frac{r * \Gamma}{t}} \tag{8}$$

We find from Formula 8 that, to guarantee that $P(q \in \mathbb{T})$ is no greater than Γ, l should be less than or equal to $\sqrt{\frac{r*\Gamma}{t}}$. Moreover, we notice that the upper bound of l increases monotonically with Γ. For instance,

- If $\Gamma = 0.01$, then $l \leq \sqrt{\frac{r*0.01}{t}} = \sqrt{\frac{r}{100*t}}$;
- If $\Gamma = 0.1$, then $l \leq \sqrt{\frac{r*0.1}{t}} = \sqrt{\frac{r}{10*t}}$.

Therefore, given a user-specified parameter Γ that defines the maximum value of $P(q \in \mathbb{T})$, the upper bound of l is $\sqrt{\frac{r*\Gamma}{t}}$.

B. MEMORY-BOUNDED SIDE-LENGTH FOR THE GRID

In subsection V-B, we analyze the maximum space needed by *GridVoronoi*, which is given in Formula 12. Let n_e, n_v be the number of edges/sides and vertices of the Voronoi Diagram, respectively. For a given dataset, n, n_e, n_v are fixed and the only variable is l, i.e. the side-length of the grid cells.

Let φ be the maximum available memory space for *GridVoronoi*, then φ should satisfy that

$$\varphi \geq 3\frac{r}{l^2} + 4n_e - 2n_v \tag{9}$$

Then we have:

$$l \geq \sqrt{\frac{3r}{\varphi - 4n_e + 2n_v}} \tag{10}$$

Therefore, the smallest (memory bounded) side-length for *GridVoronoi* is $\sqrt{\frac{3r}{\varphi-4n_e+2n_v}}$.

It should be noted that since an integer value usually takes 4 Bytes, thus in the case that φ is measured in terms of Megabytes (MB), then we shall use $\varphi * 2^{-18}$ should replace the φ in Formula 10. That is,

$$l \geq \sqrt{\frac{3r}{\varphi * 2^{-18} - 4n_e + 2n_v}} \tag{11}$$

In conclusion, given φ as the maximum memory space for *GridVoronoi*, the smallest side-length of the grid cells for *GridVoronoi* to achieve the best performance is $l = \sqrt{\frac{3r}{\varphi-4n_e+2n_v}}$.

V. THE COMPLEXITY OF GRIDVORONOI

A. TIME COMPLEXITY

Consider again the steps in Figure 2. In step (1), for building Voronoi diagrams on 2D data the worst time complexity is $O(n \log n)$ [20] and it has been generalized and improved to O(*sort*(n)) worst-case complexity in general and an O(n) complexity can be achieved for some datasets [35]. Step (2) does not take any time since the grid and squares are virtual. Step (3) takes $O(3n)$ time, for there are in total $3n-6$ edges in the Voronoi diagrams [20], and for each edge we compute the virtual squares it intersects. So the overall time complexity in the off-line pre-processing phase is $O(n \log n) + O(3n)$.

In the online query processing phase, step (4) needs only $O(1)$ time to calculate the matching virtual square for an NN query. In Step (5), lookup the number of the virtual square in the hashmap for the corresponding Voronoi cell(s) needs approximately $O(1)$ time. If the number of Voronoi cells is more than 1, we need to calculate the distances between the NN query point and the data points in these Voronoi cells; however, as discussed in Section IV, a grid cell corresponds to no more than 6 Voronoi cells on average, so we need to check very few data points to find out the final NN. Therefore, step (5) uses almost $O(1)$ time. Overall, the total time complexity of *GridVoronoi* in on-line NN query processing is nearly $O(1)$, whereas the time complexity is $O(\log n)$ for the current state of the art algorithms that rely on R-Tree-like structures for nearest neighbour search. However, we also note that, overly large grid cell length will lead to excessive query processing time, since one square unit may intersect with many Voronoi cells, can be seen in subsection VII-D. In such cases, the on-line NN query processing will not be $O(1)$.

B. SPACE COMPLEXITY

For the memory usage of *GridVoronoi*, the only concern is that we need an auxiliary hashmap structure to save the correspondences between the grid cells and the Voronoi cells. Thus it is important to analyze the space complexity of *GridVoronoi*. To keep in the hashmap the numbers of the grid cells as the keys, and the identifiers of the Voronoi cells that every grid cell intersects as the corresponding values for the keys:

1) we need $\frac{r}{l^2}$ space to keep all the numbers of the grid cells as the keys in the hashmap. This is because there are in total $\frac{r}{l^2}$ grid cells, therefore $\frac{r}{l^2}$ space is needed to keep all the numbers (i.e., identifiers) of the grid cells as the keys in the hashmap;
2) for grid cells in $\mathbb{T}$, we need up to $4n_e$ space to store in the hashmap the identifiers of the Voronoi cells that every grid cell (in $\mathbb{T}$) intersects, as the corresponding values for the grid cells. From the fact that every Voronoi edge is shared by exactly two Voronoi cells, we notice that the space needed to keep the identifiers of the Voronoi cells per Voronoi edge is twice the space occupied by the identifiers of the two Voronoi cells, which is 4. There are n_e edges, we hence need a total space of $4n_e$ to keep in the hashmap all the values for the grid cells in $\mathbb{T}$.
3) for grid cells in $\mathbb{U}$ and $\mathbb{W}$, we need up to $2(\frac{r}{l^2} - n_v)$ space to store in the hashmap the identifiers of the Voronoi cell(s) that each grid cell (in $\mathbb{U}$ or $\mathbb{W}$) intersects/contains, as the corresponding values for the grid cells. This is due to the fact that every grid cell in $\mathbb{U}$ corresponds to only Voronoi cell in the hashmap, while each grid cell in $\mathbb{W}$ corresponds to two Voronoi cells in the hashmap. Since there are in total $\frac{r}{l^2} - n_v$ grid cells that belong to $\mathbb{U}$ and $\mathbb{W}$, we hence need a maximum space of $2(\frac{r}{l^2} - n_v)$ to keep in the hashmap all the values for the grid cells in $\mathbb{U}$ and $\mathbb{W}$.

Adding the above three spaces together, we need up to $3\frac{r}{l^2} + 4n_e - 2n_v$ space to store the hashmap. Let ϖ be the space needed by the hashmap, we have:

$$\varpi \leq 3\frac{r}{l^2} + 4n_e - 2n_v \tag{12}$$

Since an integer variable commonly occupies a space of 4 Bytes, thus up to $(3\frac{r}{l^2} + 4n_e - 2n_v) * 2^{-18}$ MB space is needed to store the hashmap. We note that this is only the upper bound memory consumption of GridVoronoi; in real cases, the actual memory usage also depends on grid cell length, because a large grid cell length will make the square units intersect with many voronoi cells, whereas a small grid cell length will generate more square units and entries in the hashmap structures.

VI. PROCESSING KNN QUERIES

In this section, we discuss how to process K Nearest Neighbour (KNN) queries using *GridVoronoi*.

Given a query point q and a user-specified parameter K, which is the number of expected nearest neighbours, KNN query finds the K closest data points to q. Upon *GridVoronoi*, we propose the following algorithm to process KNN queries.

First, based upon *GridVoronoi*, we find p_0 which is the NN data point to q and insert it to the result set *RS*. Second, starting from the Voronoi cell v_0 that contains p_0, we find the Voronoi neighbours of v_0, sort them in ascending order by the distances from the data points to q and put them in a heap. Third, we remove the first data point p_1 from the heap and insert it into *RS*; next, we fetch the Voronoi neighbours of p_1 and insert them in the heap orderly. The third step loops until the number of data points in *RS* is K.

Voronoi diagram is a very effective tool in exploring the local neighbourhood of a given Voronoi cell. Therefore, once we use *GridVoronoi* to find the Voronoi cell that contains q, with the power of the Voronoi Diagram we only need to visit a limited number of neighbouring Voronoi cells to find the KNN data points.

VII. PERFORMANCE

In this section, we experimentally evaluate the proposed algorithm, *GridVoronoi*, using four real-world datasets.

Dataset A is derived from the *North East dataset* which contains 123,593 postal addresses (points) that represent three metropolitan areas of USA (New York, Philadelphia and Boston), hence three clusters. It is available at *Chorochronos*[1] (previously called "R-tree Portal"). Dataset B contains 5,922 cities and villages of Greece, obtained from the same portal *Chorochronos*. Dataset C contains populated places and cultural landmarks in US, Canada, Mexico, their mergings and divisions, a total of 24,493 points. It is freely available from the Digital Chart of the World, a comprehensive digital map of Earth.[2] Last, dataset D is obtained from the *California's Points of Interest dataset* [36][3] which contains locations of 62,556 California places.

We compare *GridVoronoi* with two well-known algorithms: *BFS* [22] and the *VoR-Tree* based algorithm [27].

We use the Boost library[4] to construct the Voronoi diagrams, which will be used in both *GridVoronoi* and *VoR-Tree*.

[1]http://www.chorochronos.org/

[2]https://worldmap.harvard.edu/data/geonode:Digital_Chart_of_the_World

[3]http://chorochronos.datastories.org/?q=node/17

[4]http://www.boost.org

For *BFS* and *VoR-Tree*, we use the R*-tree [37] implementation from *Chorochronos*. We adopt the default parameters of R*-tree.

Two different types of queries are considered in the experiments. The first set of queries corresponds to randomly-generated query points (*random queries*), while, in the second set, query points are deliberately placed either on the sides or at the vertices of the Voronoi cells (*worst queries*). Both sets of queries contain 5,000 points.

We define L as the value of the grid cell length that virtually partitions the rectangular region (plane) where a given dataset resides into 10,000 square units. Hence, we can measure the grid cell length l in units of L. E.g. $l = 2L$ (two times of L) denotes a grid cell length that partitions the same plane into 2,500 square units, whereas $\frac{L}{2}$ (half the value of L) is a grid cell length value that splits the same plane into 40,000 square units. Likewise, $\frac{L}{4}$, $\frac{L}{8}$ denotes a side-length that partitions the plane into 160,000 and 640,000 square units, respectively.

In our experiments, we assume that the queries are issued to a server with large memory, and all the nodes of the R*-tree can be loaded into memory. All algorithms are implemented in C++ and tested on a computer with an Intel Xeon E5-2690 8-core, 2.9GHz processor and 32GB Memory, running Linux Mint 17.3.

TABLE 1. Query time comparison of different methods for NN (random queries, average time, units are in ms).

dataset	size	GV	VOR	BFS	NAIVE
A	116,292	14.09	56.85	61.46	2,639.51
B	5,922	7.39	21.88	25.70	2,478.15
C	20,928	6.12	28.46	36.18	2,487.00
D	61,796	10.41	32.62	42.10	2,482.16

TABLE 2. Query time comparison of different methods for NN (worst queries, average time, units are in ms).

dataset	size	GV	VOR	BFS	NAIVE
A	116,292	14.04	64.84	90.68	2,639.45
B	5,922	8.10	24.71	32.47	2,469.74
C	20,928	7.90	36.33	49.13	2,514.81
D	61,796	27.00	46.30	60.34	2,510.75

A. EFFICIENCY ON NN QUERIES

In the first set of experiments, we evaluate the performance and the scalability of different NN algorithms. It should be noted that *GridVoronoi* uses L for the grid cell length in the experiments in Tables 1 and 2. As mentioned above, l measured in L units corresponds to the grid cell length value that virtually partitions the plane where a given dataset resides into 10,000 square units. We note that only the query processing time is reported in our experiments, not including the offline index building time for all the algorithms. The offline index building time for *GridVoronoi* is usually longer than the other algorithms, especially in the case of large datasets. Nevertheless, the index building process is done once for all, and can be used in the real-time spatial NN querying.

In Table 1, we compare the running time performance of *GridVoronoi* (GV) against *VoR-Tree*(VOR), *BFS*, and *the naive baseline NN method*, using random queries. In *the naive baseline NN method*, for each query, it always scans the whole dataset to calculate the distances from the query point to each data point, then return the data point having the smallest distance to the query point as the final result. The values reported in Table 1 correspond to average running time needed for one query (i.e. the total running time divided by the number of queries, in our experiments 5,000). It can be seen that *GridVoronoi* significantly outperforms VoR-Tree and BFS in terms of query t4ime by orders of magnitude in all 4 datasets. In specific, *GridVoronoi* is found to be 3 to 4.6 times faster than VoR-Tree, and 3.5 to 5.9 times faster than BFS. VoR-Tree is also faster than BFS in all datasets but not by a significant degree. All three algorithms are many times faster than the naive method. *GridVoronoi*, the top performing algorithm, is up to 400 times faster than the naive method.

We also find that the average running time is independent of the dataset size. For instance, dataset C contains more points than dataset B but *GridVoronoi* performs faster in C.

In Table 2, we present results of the experiments we conducted using the worst queries set. It is clear that *GridVoronoi* is significantly faster than VoR-Tree and BFS.

For worst queries, both *GridVoronoi* and VoR-Tree are slower than the case of random queries, which is expected, as these two algorithms rely on the Voronoi diagram, thus when a query is on the sides or at the vertices of the Voronoi cells they need to check 2 or more cells to find the nearest neighbour.

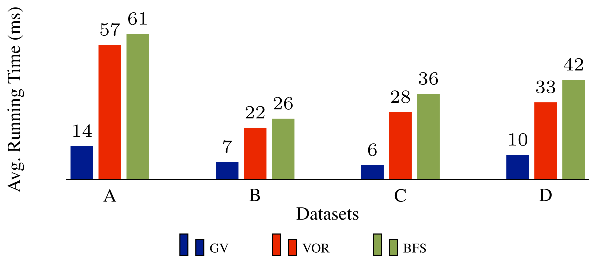

FIGURE 6. Efficiency comparison between GV, VOR, and BFS on different datasets (random queries).

These results are visualised in Figures 6 and 7. We can deduce that the performance gain depends on the data. In general, *GridVoronoi* performs significantly faster than the rest, while VoR-Tree is consistently faster than BFS. This observation is consistent with the findings in [27].

In Figure 8, we vary the grid cell length of the virtual grid to test its influence on the performance of *GridVoronoi*. As shown in the figure, when the side-length changes from $2L$ to $L/8$, the average running time decreases monotonically on every dataset for both random and worst queries.

We observe that, for the worst queries, the performance of *GridVoronoi* on dataset B and C is almost the same.

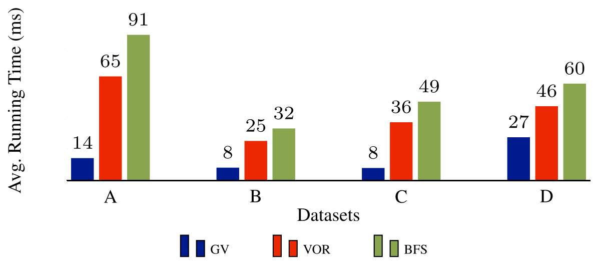

FIGURE 7. Efficiency comparison between GV, VOR, and BFS on different datasets (worst queries).

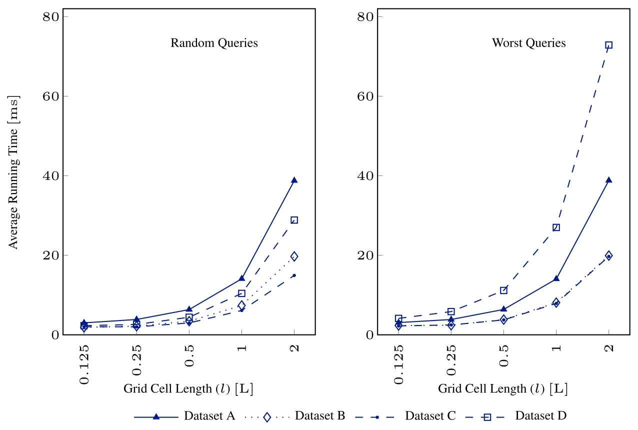

FIGURE 8. The NN query processing efficiency of *GridVoronoi* for processing NN queries on different datasets with varying grid cell length.

Also, while it is faster in random queries on dataset D than on dataset A, this is reversed for worst queries.

Using hashmap's key-value structure, for each square identifier (id) as the key, we calculate the number of elements in the corresponding value. For all the query points, we compute the sum of the number of elements for all the square ids. Then divide this sum value by the total number of query points (e.g. 5000), which we define as *the average search length*.

Average search length can measure *GridVoronoi*'s efficiency in finding the spatial NN (in the online query processing phase), since in *GridVoronoi*, once the square id of a query point is determined, we just need to check the key-value structure (in a hashmap) for this square id's corresponding value (very limited number of candidate data points), from which the final spatial NN result can be derived.

In Figure 9, we report the average search length of *GridVoronoi* in processing NN queries, with varying grid cell lengths. We see that average search length follows the same pattern as average running time, for both random and worst queries, when varying the grid cell length. This is because a smaller average search length implies shorter computation time (lookup overhead).

In Table 3, we report the relationship between the number of grid cells/squares and the value of average search length in *GridVoronoi*. We observe that, when the number of grid cells/squares is very large (i.e. when the grid cell length is

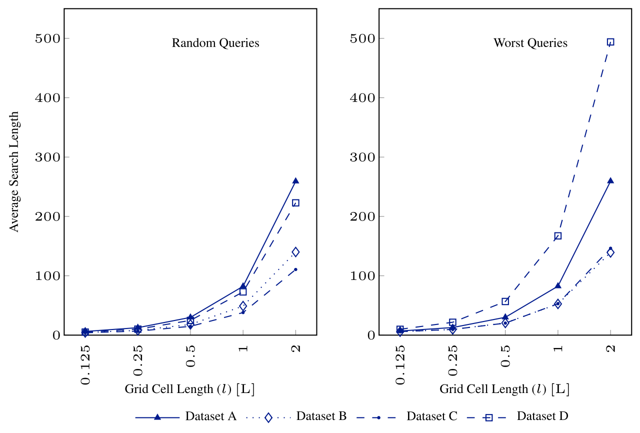

FIGURE 9. The average search length of *GridVoronoi* for processing NN queries on different datasets with varying grid cell length.

TABLE 3. Average search length for varying grid cell numbers.

# of squares	l (L)	A	B	C	D
100,000,000	0.01	2.07	2.15	2.12	2.16
25,000,000	0.02	2.25	2.30	2.26	2.36
6,250,000	0.04	2.81	2.63	2.53	2.78
1,562,500	0.08	3.40	4.29	3.14	3.76
390,625	0.16	7.93	5.33	4.66	6.28
160,000	0.25	12.56	7.97	6.78	10.11
40,000	0.50	29.91	18.03	14.83	24.82

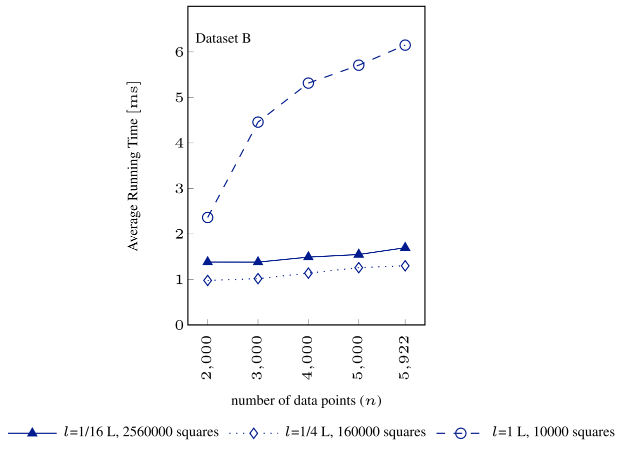

FIGURE 10. Effect of the number of data points on the running time performance of *GridVoronoi* at different grid cell lengths.

very small), *GridVoronoi* only needs to check 2−3 data points to find the spatial NN.

In Figure 10, we evaluate the scalability performance of *GridVoronoi* by measuring the average running time for 5000 queries at varying numbers of data points, with every subset of points being randomly sampled from dataset B. We see that when we are using a large number of virtual squares the average running time is almost constant, while when the total number of squares of the virtual grid is small, running time increases sharply with n, where n is the number of data points. Therefore, we can claim that the time complexity of *GridVoronoi* is almost O(1) when l is small enough. In comparison, BFS and VoR need O($\log n$) search length to find the spatial NN.

B. EFFICIENCY ON KNN QUERIES

In the second set of experiments, we compare the performance of the *GridVoronoi* based KNN algorithm, proposed in section VI, against VoR-Tree and BFS.

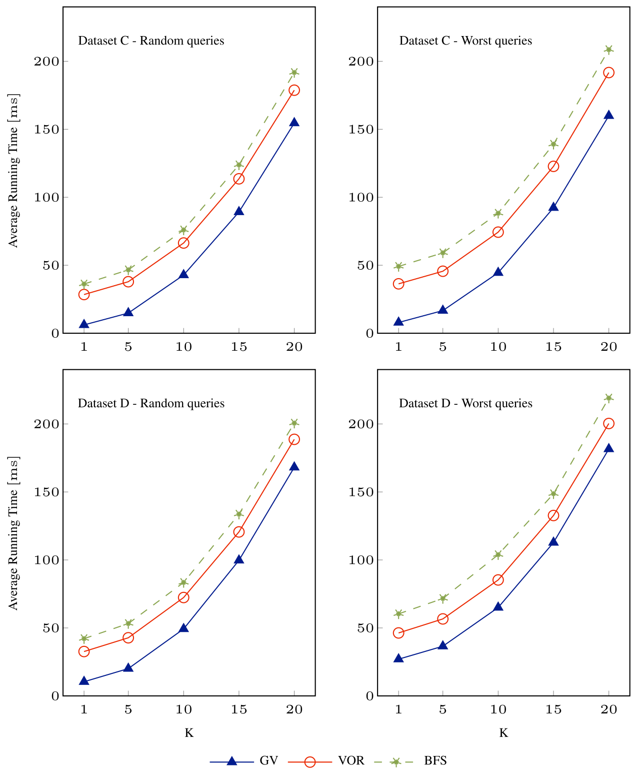

FIGURE 11. The KNN Query Processing Efficiency of Different Methods on C and D Datasets (K = 1, 5, 10, 15, 20).

In Figure 11, the 1st and 2nd plots show the performance of the three methods on dataset C, for random and worst queries respectively, when varying the number of nearest neighbours (i.e. K) from 1, 5, to 20. The grid cell length used in these experiments is $l = 1L$, in correspondence with the L value used in Figures 8 and 9.

For all the three methods, the CPU time increases as K increases; but VoR-Tree and BFS take more time than *GridVoronoi* to derive the corresponding KNN results. Moreover, we see that BFS always spends the most time, closely followed by VoR-Tree for all different K values, while *GridVoronoi* takes significantly less time than both of them. We observe that the relevant difference in running time between *GridVoronoi* and the rest slightly decreases as K increases. This is because the most time-consuming part for *GridVoronoi* is finding the NN of a query. Once the NN result is obtained, *GridVoronoi* needs to explore in the Voronoi diagram the neighbourhood of the NN result (point) to find the rest $K - 1$ nearest neighbours, which can be an iterative process, since the neighbours of the currently derived results need to be further explored (e.g., the neighbourhood of the 2nd nearest neighbour in the Voronoi diagram) to guarantee the correctness of the KNN results.

In the 3rd and 4th plots, we report the KNN query processing performance of the three methods on dataset D, for both random and worst queries. The same observations hold as with dataset C.

C. MEMORY CONSUMPTION OF GRIDVORONOI

In order to examine the influence of the grid cell length parameter L and the number of nearest neighbours K to the memory consumption of *GridVoronoi* in KNN queries we measure the memory required for the underlying hashmap structure when varying L and K. For these experiments we use 5,000 random queries, as in the subsection VII-B.

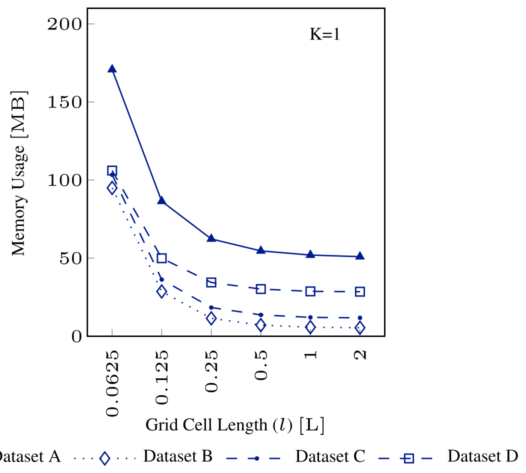

FIGURE 12. **The memory consumption of *GridVoronoi* when processing NN queries on different datasets with varying grid cell length.**

In Figure 12 we observe that, for a specific K, here $K = 1$, the memory consumption of *GridVoronoi* depends on both the size of the dataset and the grid cell length. As expected, the larger the dataset, i.e. the more data points needed to be stored in the hashmap structure, the more memory *GridVoronoi* consumes. Also, the smaller the grid cell length l, the larger memory consumption by *GridVoronoi*. It is interesting to notice that, when the value of l is lower, i.e. when we have opted for maximum running time performance, the memory consumption of *GridVoronoi* scales better, with regard to dataset size, than when l is large. For example, at $L/16$ memory usage grows from 95 MB in dataset B (5922 data points) to 170 MB in dataset A (123593 data points), i.e. a 179% increase, while at $2L$ the corresponding memory usage increase ratio is 935%.

We can infer that memory complexity follows a power law with regard to the grid cell length. The ratio of the memory usage values between two datasets increases monotonically with the ratio of the sizes of these two datasets, asymptotically becoming equal when l tends to infinity. The memory complexity increases exponentially with l but by a lower order than 2, which was the theoretical value in equation 12 after the analysis in subsection V-B.

In Figure 13, we compare the memory usage of *GridVoronoi* (GV) with different grid cell lengths, VoR-Tree, and BFS algorithms, for KNN query processing on datasets C and D at different values of K. It is clear that *GridVoronoi* consumes significantly less memory than the two other algorithms. Furthermore, we observe that, given a specific grid cell length value, the memory consumption of *GridVoronoi* is independent of K when K is 5 or larger. The memory consumption of BFS and VoR-Tree resembles that of *GridVoronoi*; they are also independent of K when $K \geq 5$.

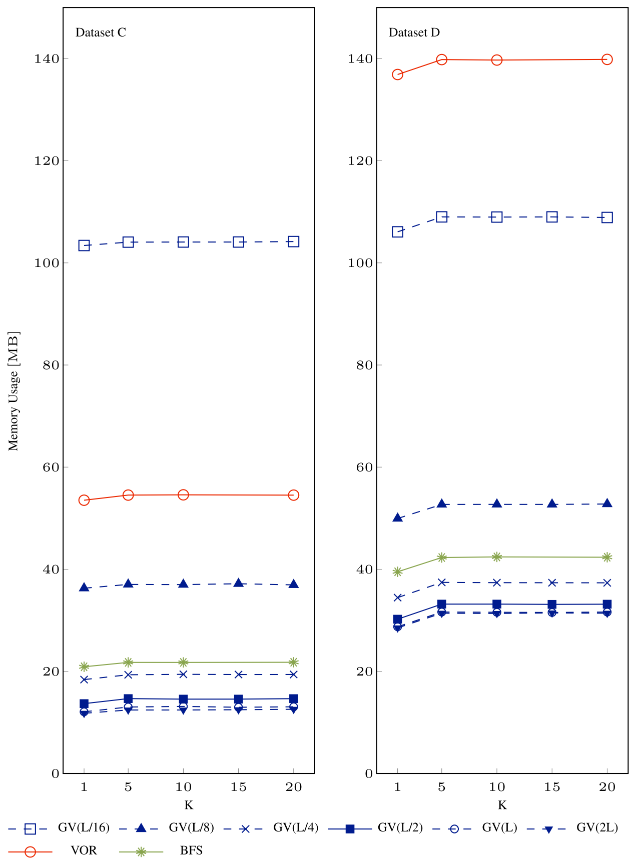

FIGURE 13. **The memory consumption of *GridVoronoi* (using different grid cell lengths), VoR-Tree, and BFS for processing KNN queries on datasets C and D with varying *K*.**

From Figure 13, we observe that VoR-Tree is highly sensitive to the size of the dataset. For dataset D, which has 3 times the number of data points than dataset C, the corresponding memory usage for VoR-Tree is 2.6 times higher. This outcome is to be expected, as the R-Tree spatial indexing hierarchical data structure is more appropriate for on-disk rather than in-memory approaches.

BFS is significantly more memory efficient than VoR-Tree, consuming 2.5 to 3.5 times less memory. *GridVoronoi* is the clear winner when grid cell length is set to be $L/4$ or larger. As we have seen in subsection VII-B (see Figure 11), *GridVoronoi* outperforms VoR-Tree and BFS, in terms of average (query processing) running time when the grid cell length is $1L$, while we have shown that average running time performance for *GridVoronoi* increases monotonically as the grid cell length gets smaller (see Figure 8). We can deduce that *GridVoronoi* is superior to both VoR-Tree and BFS, in terms of running time performance and memory consumption, when the grid cell length l is between $L/4$ and $1L$.

In summary, we find that there is a trade-off between running time efficiency and memory footprint, which, in the case of *GridVoronoi*, can be managed by the grid cell length parameter l. Overall, the lookup efficiency of *GridVoronoi* is almost O(1), hence it can significantly speed up spatial nearest neighbour query processing.

D. LARGE-SCALE EXPERIMENTS

In order to further investigate the scalability of *GridVoronoi* on large datasets, we use *OpenStreetMap Data Extracts*,[5] which are updated daily and contain POIs from different continents. In our experiments, we use the Asia and Europe datasets, which contain 4,002,125 and 9,199,696 data points, respectively. We note that the sizes of the Asia and Europe datasets are 32-79 times of that of dataset A (*North East dataset*), which was used in subsection VII-A.

In order to check the influence of the number of data points on the memory consumption and time efficiency, we gradually increase the number of data points from the same dataset. For instance, on the Europe dataset, which has a total of 9,199,696 data points, but we only use the first 1,000,000 data points in the beginning for the experiments, then gradually increase this number to 9,000,000, with a step size of 1,000,000. In the experiments, we generate 5,000 random query points (which are spatial NN queries) for each dataset and fix the query points for all the experiments on the same dataset. In order to eliminate systematic errors in measurement due to CPU caching or I/O overhead, we run each experiment three times and introduce a 3 seconds timeout between two experiments. Since each experiment will run three times, we use MAD (Median Absolute Deviation) to handle possible outliers present so that the reported results will be more robust.

In Figures 14 and 15, we present the memory consumption (units are in MB) of *GridVoronoi* on the Asia and Europe dataset at varying number of data points and different granularities of grid cell length. In Figure 15, we observe that, for the same number of data points, memory usage does not vary significantly with grid cell length, except at $l = L/16$ where *GridVoronoi* consumes significantly more memory than the rest. On the other hand, for the same grid cell length, memory usage increases linearly with the number of data points. Similar observations also hold for the other dataset, i.e. Asia, as shown in Figure 14.

In Tables 4 and 5, we report the raw values of query processing time (units are in seconds)and the total running time (which includes both the off-line index building time and the on-line query processing time) of *GridVoronoi* on the Asia dataset, with varying number of data points and different

[5]https://download.geofabrik.de, accessed on 2019-06-18.

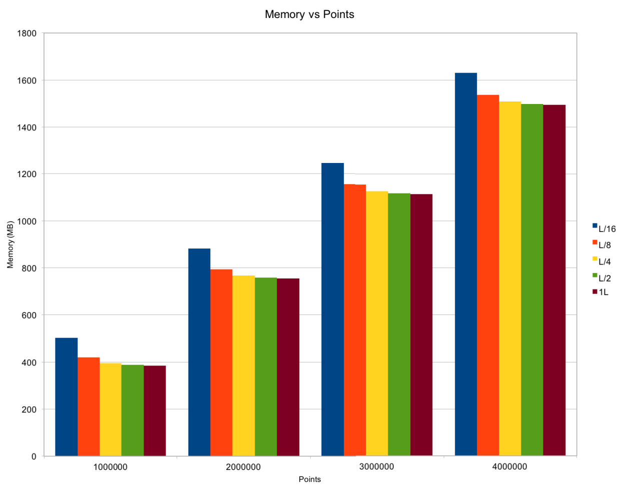

FIGURE 14. Memory consumption of *GridVoronoi* on the Asia dataset with varying number of data points and different granularities of grid cell length.

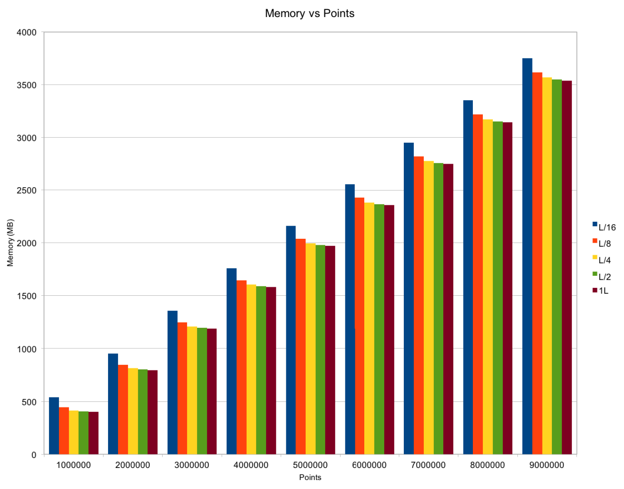

FIGURE 15. Memory consumption of *GridVoronoi* on the Europe dataset with varying number of data points and different granularities of grid cell length.

TABLE 4. Query processing time of GridVoronoi on the Asia dataset with varying grid cell lengths (units are in seconds).

points	L/16	L/8	L/4	L/2	1L
1,000,000	3.31	3.63	4.90	9.58	20.71
2,000,000	3.54	4.34	7.32	17.89	44.75
3,000,000	3.98	4.97	9.41	25.45	64.48
4,000,000	3.99	5.65	11.58	33.19	86.18

TABLE 5. Total running time of GridVoronoi on the Asia dataset with varying grid cell lengths (units are in seconds).

points	L/16	L/8	L/4	L/2	1L
1,000,000	23.84	19.11	21.13	28.10	42.55
2,000,000	40.88	40.01	47.18	67.96	109.29
3,000,000	59.61	62.01	80.55	122.50	201.34
4,000,000	78.68	89.74	120.21	189.41	320.79

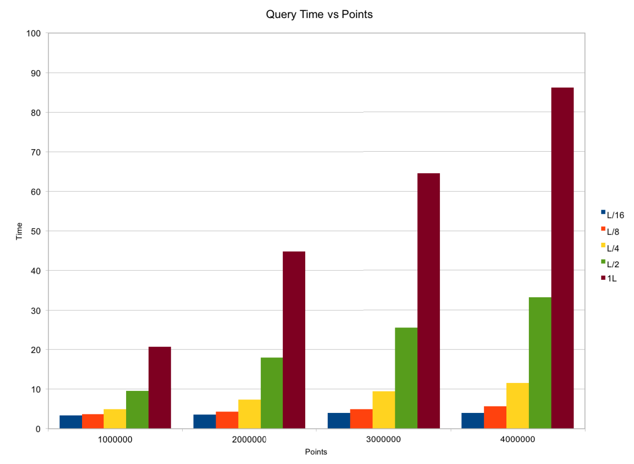

FIGURE 16. Query processing time of *GridVoronoi* on the Asia dataset with varying number of data points.

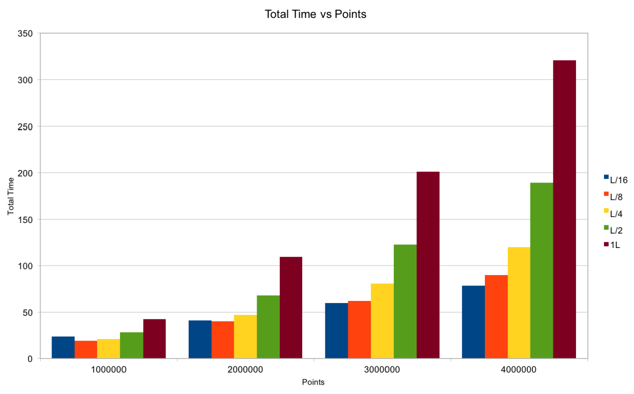

FIGURE 17. The total running time of *GridVoronoi* on the Asia dataset with varying number of data points.

TABLE 6. Index building time of *GridVoronoi* on the Asia dataset with varying grid cell lengths (units are in seconds).

points	L/16	L/8	L/4	L/2	1L
1,000,000	20.53	15.48	16.23	18.52	21.84
2,000,000	37.34	35.67	39.86	50.07	64.54
3,000,000	55.63	57.04	71.14	97.05	136.86
4,000,000	74.69	84.09	108.63	156.22	234.61

grid cell lengths. Moreover, we visualize the results in Figures 16 and 17, respectively. In terms of query processing time, we see that it is generally anti-monotonic with grid cell length, thus *GridVoronoi* obtains the best time efficiency at $l = L/16$. In Table 6, we also report the index building time, by subtracting the corresponding values in the above two tables. It is interesting to observe that the index building time of *GridVoronoi* does not decrease with larger grid cell length values; on the contrary, it usually increases with grid cell length. This is due to the reason that there will be more intersections between the squares and the Voronoi cells when the grid cell length becomes larger. However, smaller grid cell length also brings more square units and more entries in the hashmap structures (thus occupies more memory space). Hence, there should be an internal balance between the memory space required by the intersections between the squares and the Voronoi cells, and the extra memory space needed by the square units and the hashmap structures. This can also explain the two exception cases in index building time between $l = L/16$ and $l = L/8$ in the first two rows of Table 6, where $l = L/8$ takes less time in building the index. This also indicates that, the space assigned to the square units (the keys of the hashmap structures) is not expensive even when the grid cell length is small, and the most memory-occupying structures are the values of the hashmap structures and the data points of the dataset.

On the other hand, both query processing time and index building time increases monotonically with the number of data points, as can be observed in Tables 4 and 6. The larger the number of data points, the bigger the gap in both query processing and index building time among different grid cell lengths.

TABLE 7. Query processing time of *GridVoronoi* on the Europe dataset with varying grid cell lengths (units are in seconds).

points	L/16	L/8	L/4	L/2	1L
1,000,000	3.80	4.19	5.74	12.47	30.21
2,000,000	4.08	5.27	8.83	21.49	57.21
3,000,000	4.49	5.96	11.74	31.13	84.94
4,000,000	4.71	6.78	14.48	39.11	110.67
5,000,000	4.97	7.91	16.37	48.77	139.80
6,000,000	5.21	8.41	18.80	55.54	167.45
7,000,000	5.48	8.93	21.25	62.80	193.30
8,000,000	5.74	10.09	23.63	72.66	222.01
9,000,000	6.00	10.74	26.64	83.07	250.13

TABLE 8. Total running time of *GridVoronoi* on the Europe dataset with varying grid cell lengths (units are in seconds).

points	L/16	L/8	L/4	L/2	1L
1,000,000	24.59	21.27	20.02	24.75	35.76
2,000,000	44.18	42.29	43.71	51.99	76.93
3,000,000	64.60	63.07	66.68	85.47	131.17
4,000,000	84.52	81.33	96.92	113.29	202.56
5,000,000	104.44	108.10	122.01	171.02	285.74
6,000,000	126.90	130.54	156.78	207.93	367.14
7,000,000	148.41	150.73	182.61	251.69	455.34
8,000,000	168.19	178.94	222.07	311.33	579.07
9,000,000	185.60	206.86	257.37	374.58	695.29

In Tables 7 and 8, and Figures 18 and 19, we report the query processing time and total running time of *GridVoronoi* on the Europe dataset, where we have similar observations as the Asia dataset.

Finally, we find that the time efficiency of *GridVoronoi* using a grid cell length of $l = L/8$ is close to that of $l = L/16$, but the former occupies significantly less memory than the former, as can be seen from Figures 14 and 15.

E. INFLUENCE OF POINT DENSITY OF A DATASET ON QUERY EFFICIENCY

In the above subsection, we vary the number of data points from the same dataset to check the influence of the number of data points on the efficiency of query processing. Since the data points in the Asia and Europe datasets are not ordered

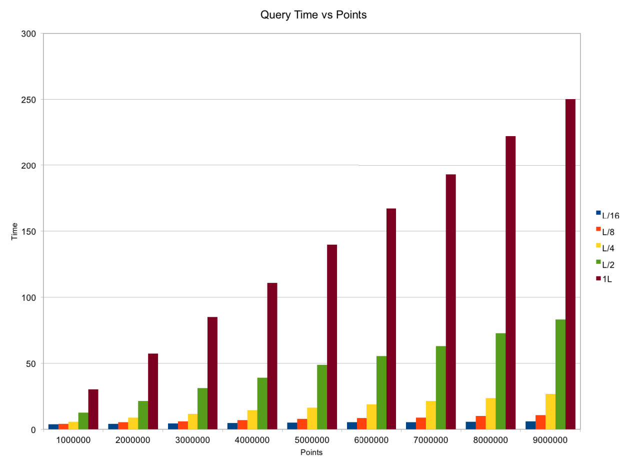

FIGURE 18. Query processing time of *GridVoronoi* on the Europe dataset with varying number of data points.

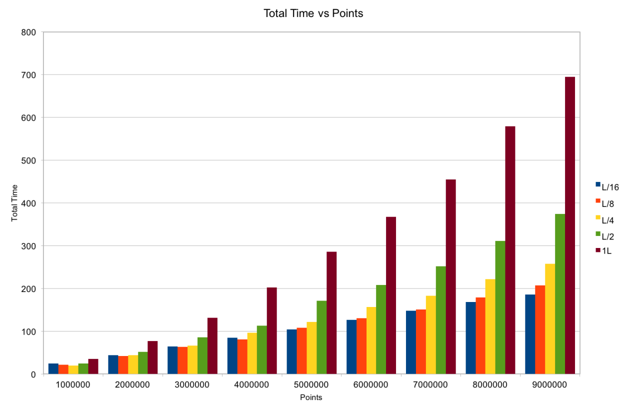

FIGURE 19. The total running time of *GridVoronoi* on the Europe dataset with varying number of data points.

by their coordinates in advance, the point density of the dataset gradually will also become larger when we increase the number of data points with a scale of 1,000,000 data points. Especially for the Europe dataset, its point density progressively increases when the number of data points which varies from 1,000,000 to 9,000,000, with a total of 9 steps.

On the Europe dataset, it is clear that the total running time of *GridVoronoi* increases almost linearly with the number of data points (when we fix the grid cell length), as can be seen from Table 8 and Figure 19. Similar pattern can be observed on the Asia dataset, as can be seen from Table 5 and Figure 17. In terms of query processing time, it also increases when the point density of the dataset increases, which can be observed from Tables 4 and 7.

In summary, for the same grid cell length, both query processing time and total running time increase monotonically with the point density of the datasets.

F. SHORTCOMINGS OF GRIDVORONOI

The proposed spatial index *GridVoronoi* has a few shortcomings that we should point out.

1) *GridVoronoi* is a combination of a virtual grid of squares and the Voronoi diagram structure. It relies on the latter to split the 2D plane into non-overlapping Voronoi cells. However, when the data points are inserted or deleted, we need to run Voronoi diagram from scratch to re-split the 2D plane, since we do not have a local update technique for Voronoi diagram. The same issue exists for all the algorithms that rely on Voronoi diagram.
2) A skewed distribution of data points in a dataset might cause low query processing efficiency in the "dense" regions. This is because there will be more intersections between the square units and the Voronoi cells in the dense regions. In Table 2 and Figure 8 in subsection VII-A, we present the results with worst queries, which reside on the edges and vertices of the voronoi cells. For small grid cell length(e.g. $l = L/8$), we infer that the effect of skewness on query processing is small; but for large grid cell lengths, the increase in running time (due to data skewness) will be significant. In fact, in the Asia and Europe datasets, there exist many dense regions (e.g. cities) of POIs, and many sparse regions (suburbs and fields).
3) *GridVoronoi* consumes excessively more memory space than the other methods. For instance, on the Europe dataset, it occupies nearly 4 GB memory space. This could be tolerated/accepted for applications that give priority to time efficiency and are equipped with high performance computing facilities.
4) *GridVoronoi* requires a huge amount of time for the off-line index building. Once the index is built, it can be used for on-line query processing, to process vast amount of real-time user queries.
5) *GridVoronoi* currently only applies to 2D datasets.

VIII. CONCLUSION AND FUTURE WORK

This paper approaches an important problem in spatial query processing, i.e., spatial nearest neighbour search, in a practical way. We present *GridVoronoi* which is a Voronoi Diagram complemented by a virtual grid to make the NN search efficient. It adopts the virtual grid for promptly finding the grid cell where the query point locates, then employs Voronoi Diagram which is highly efficient in exploring the local neighbourhood of the corresponding voronoi cell(s) that contain(s)/intersect(s) the grid cell for spatial nearest neighbour. Experiments on four real-world data sets have confirmed the efficiency and effectiveness of *GridVoronoi*. Through large-scale experiments, we show that the time efficiency of *GridVoronoi* depends both on the number of data points and the grid cell length adopted.

In future work, we will investigate how to extend *GridVoronoi* to 3D space where the height dimension will be taken into consideration in spatial NN querying. We will also extend our techniques to road networks, where we will use the virtual grid to partition the 2D space; then calculate the intersections/correspondences between the square units and road segments, and store them in the hashmap-like structures for fast NN processing over the road networks.

ACKNOWLEDGMENT

The first author would like to thank Prof. Heri Ramampiaro of Norwegian University of Science and Technology (NTNU, Norway) for the insightful discussions on this work, during the first author's stay at NTNU as an ERCIM Marie-Curie Fellow. The authors would like to thank the anonymous reviewers who have provided constructive suggestions that are very helpful to this work.

REFERENCES

[1] W. Alsabhan and O. B. Ali, "Mobile land information system (MLIS): A GIS-based e-government application," *Int. J. Mobile Learn. Org.*, vol. 6, nos. 3–4, pp. 260–284, 2012.

[2] Y. Shi, T. Boudouh, and O. Grunder, "A robust optimization for a home health care routing and scheduling problem with consideration of uncertain travel and service times," *Transp. Res. E, Logistics Transp. Rev.*, vol. 128, pp. 52–95, Aug. 2019.

[3] Y. Lai, Z. Lv, K.-C. Li, and M. Liao, "Urban traffic Coulomb's law: A new approach for taxi route recommendation," *IEEE Trans. Intell. Transp. Syst.*, vol. 20, no. 8, pp. 3024–3037, Aug. 2019. doi: 10.1109/TITS.2018.2870990.

[4] N. Alam and A. G. Dempster, "Cooperative positioning for vehicular networks: Facts and future," *IEEE Trans. Intell. Transp. Syst.*, vol. 14, no. 4, pp. 1708–1717, Dec. 2013.

[5] T. Wang, J. Zhou, A. Liu, M. Bhuiyan, G. Wang, and W. Jia, "Fog-based computing and storage offloading for data synchronization in IoT," *IEEE Internet Things J.*, vol. 6, no. 3, pp. 4272–4282, 2019.

[6] T. Wang, G. Zhang, A. Liu, M. Z. A. Bhuiyan, and Q. Jin, "A secure IoT service architecture with an efficient balance dynamics based on cloud and edge computing," *IEEE Internet Things J.*, vol. 6, no. 3, pp. 4831–4843, Jun. 2019.

[7] K. Ren, W. Lou, and Y. Zhang, "LEDS: Providing location-aware end-to-end data security in wireless sensor networks," *IEEE Trans. Mobile Comput.*, vol. 7, no. 5, pp. 585–598, May 2008.

[8] G. Yan, S. Olariu, and M. C. Weigle, "Providing location security in vehicular Ad Hoc networks," *IEEE Wireless Commun.*, vol. 16, no. 6, pp. 48–55, Dec. 2009.

[9] W. Wang, Y. Li, X. Wang, J. Liu, and X. Zhang, "Detecting Android malicious apps and categorizing benign apps with ensemble of classifiers," *Future Gener. Comput. Syst.*, vol. 78, pp. 987–994, Jan. 2018.

[10] X. Liu, J. Liu, S. Zhu, W. Wang, and X. Zhang, "Privacy risk analysis and mitigation of analytics libraries in the Android ecosystem," *IEEE Trans. Mobile Comput.*, to be published.

[11] Y. Huang, J. Pei, and H. Xiong, "Mining co-location patterns with rare events from spatial data sets," *Geoinformatica*, vol. 10, no. 3, pp. 239–260, 2006.

[12] M. Khan, Q. Ding, and W. Perrizo, "K-nearest neighbor classification on spatial data streams using p-trees," in *Proc. 6th Pacific–Asia Conf. Adv. Knowl. Discovery Data Mining*, 2002, pp. 517–518.

[13] C. Zhang, C. Liu, X. Zhang, and G. Almpanidis, "An up-to-date comparison of state-of-the-art classification algorithms," *Expert Syst. Appl.*, vol. 82, pp. 128–150, Oct. 2017.

[14] J. Bi and C. Zhang, "An empirical comparison on state-of-the-art multi-class imbalance learning algorithms and a new diversified ensemble learning scheme," *Knowl.-Based Syst.*, vol. 158, pp. 81–93, Oct. 2018.

[15] T. N. Alotaiby, S. A. Alshebeili, L. M. Aljafar, and W. M. Alsabhan, "ECG-based subject identification using common spatial pattern and SVM," *J. Sensors*, vol. 2019, Mar. 2019, Art. no. 8934905.

[16] J. B. Rocha-Junior, O. Gkorgkas, S. Jonassen, and K. Nørvåg, "Efficient processing of top-k spatial keyword queries," in *Proc. SSTD*, 2011, pp. 205–222.

[17] M. L. Yiu, X. Dai, N. Mamoulis, and M. Vaitis, "Top-k spatial preference queries," in *Proc. IEEE ICDE*, Apr. 2007, pp. 1076–1085.

[18] J. Bao, Y. Zheng, and M. F. Mokbel, "Location-based and preference-aware recommendation using sparse geo-social networking data," in *Proc. 20th Int. Conf. Adv. Geograph. Inf. Syst. (SIGSPATIAL)*, 2012, pp. 199–208.

[19] D. Wu, M. L. Yiu, G. Cong, and C. S. Jensen, "Joint top-k spatial keyword query processing," *IEEE Trans. Knowl. Data Eng.*, vol. 24, no. 10, pp. 1889–1903, Jul. 2012.

[20] F. Aurenhammer, "Voronoi diagrams—A survey of a fundamental geometric data structure," *ACM Comput. Surv.*, vol. 23, no. 3, pp. 345–405, 1991.

[21] A. Okabe, B. Boots, K. Sugihara, and S. N. Chiu, *Spatial Tessellations: Concepts and Applications of Voronoi Diagrams*, vol. 501. Hoboken, NJ, USA: Wiley, 2009.

[22] G. R. Hjaltason and Hanan Samet, "Distance browsing in spatial databases," *ACM Trans. Database Syst.*, vol. 24, no. 2, pp. 265–318, Jun. 1999.

[23] L. Hu, W.-S. Ku, S. Bakiras, and C. Shahabi, "Verifying spatial queries using Voronoi neighbors," in *Proc. 18th SIGSPATIAL Int. Conf. Adv. Geograph. Inf. Syst.*, 2010, pp. 350–359.

[24] I. D. Felipe, V. Hristidis, and N. Rishe, "Keyword search on spatial databases," in *Proc. IEEE 24th Int. Conf. Data Eng. (ICDE)*, Apr. 2008, pp. 656–665.

[25] G. Cong, C. S. Jensen, and D. Wu, "Efficient retrieval of the top-k most relevant spatial Web objects," *Proc. VLDB Endowment*, vol. 2, no. 1, pp. 337–348, 2009.

[26] X. Cao, G. Cong, S. Christian Jensen, and B. C. Ooi, "Collective spatial keyword querying," in *Proc. SIGMOD Conf.*, 2011, pp. 373–384.

[27] M. Sharifzadeh and C. Shahabi, "Vor-tree: R-trees with Voronoi diagrams for efficient processing of spatial nearest neighbor queries," *Proc. VLDB Endowment*, vol. 3, nos. 1–2, pp. 1231–1242, 2010.

[28] S. Nutanong, R. Zhang, E. Tanin, and L. Kulik, "The v*-diagram: A query-dependent approach to moving KNN queries," *Proc. VLDB Endowment*, vol. 1, no. 1, pp. 1095–1106, 2008.

[29] C. Li, Y. Gu, J. Qi, G. Yu, R. Zhang, and W. Yi, "Processing moving KNN queries using influential neighbor sets," *Proc. VLDB Endowment*, vol. 8, no. 2, pp. 113–124, 2014.

[30] K. Mouratidis, D. Papadias, and M. Hadjieleftheriou, "Conceptual partitioning: An efficient method for continuous nearest neighbor monitoring," in *Proc. ACM SIGMOD Int. Conf. Manage. Data SIGMOD*, 2005, pp. 634–645.

[31] C. Ji, Z. Li, W. Qu, Y. Xu, and Y. Li, "Scalable nearest neighbor query processing based on inverted grid index," *J. Netw. Comput. Appl.*, vol. 44, pp. 172–182, Sep. 2014.

[32] Y. Gu, G. Yu, and X. Yu, "An efficient method for k nearest neighbor searching in obstructed spatial databases," *J. Inf. Sci. Eng.*, vol. 30, no. 5, pp. 1569–1583, 2014.

[33] B. Zheng, J. Xu, W.-C. Lee, and D. L. Lee, "Grid-partition index: A hybrid method for nearest-neighbor queries in wireless location-based services," *VLDB J.*, vol. 15, no. 1, pp. 21–39, 2006.

[34] C.-Y. Chow, M. Mokbel, J. Naps, and S. Nath, "Approximate evaluation of range nearest neighbor queries with quality guarantee," in *Advances in Spatial and Temporal Databases*. Berlin, Germany: Springer-Verlag, 2009, pp. 283–301.

[35] K. Buchin and W. Mulzer, "Delaunay triangulations in O(sort (n)) time and more," *J. ACM*, vol. 58, no. 2, p. 6, 2011.

[36] F. Li, D. Cheng, M. Hadjieleftheriou, G. Kollios, and S.-H. Teng, "On trip planning queries in spatial databases," in *Proc. 9th Int. Conf. Adv. Spatial Temporal Databases (SSTD)*, 2005, pp. 273–290.

[37] N. Beckmann, H.-P. Kriegel, R. Schneider, and B. Seeger, "The R*-tree: An efficient and robust access method for points and rectangles," in *Proc. ACM SIGMOD Int. Conf. Manage. Data (SIGMOD)*, 1990, pp. 322–331.

CHONGSHENG ZHANG received the Ph.D. degree from INRIA, France. From 2012 to 2013, he was an ERCIM Marie-Curie Fellow with the Norwegian University of Science and Technology, Norway. He is currently a Professor with Henan University and the Director of big data research. He has published more than 20 articles in peer-reviewed journals and conferences, including *Information Sciences*, *Expert Systems With Applications*, *Knowledge-Based Systems*, and the IEEE ICDM 2010. He has authored three books and holds six Chinese patents. His research interests include databases and machine learning.

GEORGE ALMPANIDIS received the B.Sc. degree in physics from the Aristotle University of Thessaloniki, the M.Sc. degree in information technology from Glasgow University, and the Ph.D. degree in computing science from the Aristotle University of Thessaloniki. He has been an ERCIM Marie-Curie Fellow with the Norwegian University of Science and Technology and a Postdoctoral Researcher with the National University of Ireland. He is currently a Professor with Henan University. His research interests include machine learning, information retrieval, and speech and language processing.

FAEGHEH HASIBI received the Ph.D. degree in computer science from the Norwegian University of Science and Technology (NTNU). She is currently a Tenure-Track Assistant Professor with the Institute of Computing and Information Sciences (iCIS), Radboud University, The Netherlands. She has published many articles in information retrieval and holds multiple patents. Her research interests include semantic information retrieval and machine learning. She received multiple awards.

GAOJUAN FAN received the Ph.D. degree from the Nanjing University of Posts and Telecommunications (NJUPT). She is currently an Associate Professor with Henan University. Her research interests include the Internet of Things (IoT) and wireless sensor networks (WSNs).