ASTES
Advances in Science, Technology and Engineering Systems Journal Vol. 5, No. 4, 160-166 (2020)
www.astesj.com
Special Issue on Multidisciplinary Innovation in Engineering Science & Technology
ASTESJ
ISSN: 2415-6698

Nearest Neighbour Search in k-dSLst Tree

Meenakshi Hooda[*], Sumeet Gill

Department of Mathematics, Maharshi Dayanand University, 124001, India

ARTICLE INFO
Article history:
Received: 12 May, 2020
Accepted: 26 June, 2020
Online: 18 July, 2020

Keywords:
Nearest Neighbour
Spatial Indexing
k-d tree
Sorted Linked List
Duplicate Keys

ABSTRACT
In the last few years of research and innovations, lots of spatial data in the form of points, lines, polygons and circles have been made available. Traditional indexing methods are not perfect to store spatial data. To search for nearest neighbour is one of the challenges in different fields like spatiotemporal data mining, computer vision, traffic management and machine learning. Many novel data structures are proposed in the past, which use spatial partitioning and recursive breakdown of hyperplane to find the nearest neighbour efficiently. In this paper, we have adopted the same strategy and proposed a nearest neighbour search algorithm for k-dSLst tree. k-dSLst tree is based on k-d tree and sorted linked list to handle spatial data with duplicate keys, which is ignored by most of the spatial indexing structures based on k-d tree. The research work in this paper shows experimentally that where the time taken by brute force nearest neighbour search increases exponentially with increase in number of records with duplicate keys and size of dataset, the proposed algorithm k-dSLstNearestNeighbourSearch based on k-dSLst tree performs far better with approximately linear increase in search time.

1. Introduction

Geospatial datasets are huge and complex in structure and relationships. We need complex spatial operators to fetch the required information from spatial datasets. These complex spatial operators include intersection, overlap, adjacency etc. The traditional indexing structures can't handle queries related to spatial details efficiently. Spatial queries use spatial relationships among different geometries and make use of n-dimensional geometric data such as points, lines and polygons to retrieve required data.

A spatial query retrieves features depending on relationships of spatial data with geometry of queried data. The objective of spatial queries is to find out the spatial relationships in one or more subjects to search for spatial objects. The extracted information help in taking decisions related to various policies or for doing analysis in various fields. Spatial queries permit for the utilization of data types related to geometry like point, line etc. and take into consideration the spatial relationship in these geometries. Spatial indices are used for spatial database to optimize spatial query. The indices provide one of the optimization techniques for improving the quality of services based on location.

1.1 Range Query

The range queries are related with bounded areas. The output of such kind of queries contains some region that might be overlapped. The spatial objects are associated to each other within specific area or distance. These kinds of queries have associated area and require at least two parameters i.e. location and boundary limit.

Example:

Find all the hostels within 12 kilometers of a given university.

Find all towns within 90 kilometers of a given village.

1.2 Spatial Join Query

These queries are the combination of more than one spatial query. We need spatial join operation for retrieving required information. These queries are expensive as join condition(s) involve areas and their proximity to each other.

Example:

Find all the cities near given lake.

[*]Meenakshi Hooda, Maharshi Dayanand University, Haryana, India & mshthebest@gmail.com

The partitioning and summarization of spatiotemporal data is significant for numerous community based applications like environmental science, public safety and health [1]. The relations among spatial objects can be categorized in different ways such as classifications based on distance, topology and direction relations. These may be further joined using logical operators to depict neighbourhood relation between spatial objects [2].

2. Related Work

2.1 Nearest Neighbour Query

Nearest Neighbour Search (NNS) is also called as Proximity, Similarity or Closest Point Search. It is an optimization problem to find nearest points in metric spaces [3]. In such kind of queries, the data closet to the queried location is retrieved. Here, we search for objects which are nearest to the particular location. We can also query for xNN i.e. x nearest neighbours, in which search is done for x objects nearest to the queried location. The result might be ordered by their respective proximity to queried spatial location. [4] introduce a method to fetch nearest neighbour data in 3-Dimensional space by making use of clustered hierarchical indexing tree structure. Studies show that this approach achieves remarkable improvement in response time analysis as compared to already existing spatial data accessing methods in databases. [5] research work introduces a progressive algorithm for a search of approximate k-nearest neighbor. The most of the KNN algorithms though utilize k-nearest neighbor libraries for many of the data analysis procedures, but the fact is that these algorithms run only after indexing of the whole dataset, which means that the datasets are not online. [6] proposed a method of parallel kd-tree construction for 3-D points on a Graphic Processing Unit. The method consists of a sorting algorithm to maintain high level of parallelism throughout the creation.

Example: Find 5 nearest hostels with respect to given location comprised of n-dimensional coordinate.

In this research paper, we are introducing the algorithm k-dSLstTreeNearest for Nearest Neighbour in k-dSLst tree structure designed in our previous work to index spatial data with duplicate keys.

3. Brute Force Method: Nearest Neighbour Search for Spatial Data with Duplicate Keys

Brute Force exploration process, which is also called as Exhaustive/Blind Search, is extremely generalized technique to solve any problem. It systematically enumerates all possible contenders for the problem's solution and tests to check if the contender fulfills the statement of the problem [7]. In this algorithm, we find the distance of every object in dataset from the position of the queried object. As, multiple objects can be found at the nearest neighbour location, a list is maintained to keep record of all objects with minimum distance. Algorithm 1, **Distance bruteForceNearestNeighbourSearch (SpatialDataset, Location, ResultNN_BF *resultNearestNeighbourBF)**, receives *spDataSet* of *SpatialDataSet* type, k dimensional *queryPosition* of *Location* type to which nearest neighbour is to be found and returns either *INFINITY*, if no record is found in dataset or minimum distance of the nearest found object(s) from *queryPosition* in case record(s) is found. Also, a pointer *resultNearestNeighbourBF* of *ResultNN_BF* type is passed to save the address of result node(s), if found.

If there is no record in dataset and list is empty i.e. *spDataSet* is NULL, **bruteForceNearestNeighbourSearch** will return INFINITY. On the other hand, if records exist in the list, initialize *minDistance* to INFINITY. Now, traverse every node in the *spDataSet* to find the distance between *queryPosition* and *currentRecord*. Also, update the minimum distance *minDistance*, if it is lesser than the *minDistance* calculated up to now. Now, insert object details in *resultNearestNeighbourBF* which satisfy the minimum distance condition.

Algorithm 1 bruteForceNearestNeighbourSearch

```
1:   Begin
2:   if spDataSet is NULL
3:       return INFINITY
4:   end if
5:   Initialize minDistance to INFINITY
6:   Set distanceSqr to 0
7:   Set currentRecord to spDataSet
8:   loop while currentRecord is NOT NULL
9:       Find distanceSqr between currentRecord and
         queryPosition
10:      Insert calculated distaceSqr to currentRecord
         temporarily
11:      if distanceSqr < minDistance
12:      then
13:          minDistance = distanceSqr
14:      end if
15:      Move to next currentRecord
16:      Reset distanceSqr to 0
17:  end loop
18:  Set currentRecord to spDataSet
19:  loop while currentRecord exist
20:      if distanceSqr inserted to currentRecord temporarily ==
         minDistance
21:      then
22:          add currentRecord to resultNearestNeighbourBF
23:      end if
24:      Move to next currentRecord
25:  end Loop
26:  return minDistance
27:  End
```

4. k-dSLst Tree

In our earlier research work, we had introduced an indexing structure k-dSLst tree to index spatial data with duplicate keys. k-dSLst tree is based on k-d tree and sorted linked list. In this paper, we are introducing an algorithm to find nearest neighbour for a given n-dimensional spatial point in a spatial dataset. In k-dSLst tree, first a *kdSLstNode* is created according to n-dimensional spatial location of an object, and then we insert the data related to same object in *dataSNode* in *kdSLstNode* created above. If kdSLstNode related to n-dimensional point already exists, then *dataSNode* is inserted in a sorted way according to *object_Id* of the object in already existing *kdSLstNode*. When we insert the *dataSNode*, we need to find the right location for insertion by traversing the linked list at particular *kdSLstNode*. It will increase the insertion time as compared to k-dLst tree, but when comes to searching for a particular object at particular location, it will outperform k-dLst tree on an average.

Figure 1 shows the structure of 2-dSLst tree for dataset records in 2-d space stored as *kdSLstNodes* in a 2-dSLst tree, which can be generalized for k-d space to create k-dSLst tree. A rectangular space is also represented by the leaf *kdSLstNode* which is further separated into two spaces by the newly inserted point.

The root *kdSLstNode* will have discriminator 0. It will be 1 for two sons at the next level, and will be incremented for every next level until it reaches up to *k-1* on k^{th} level. Then it again starts with 0 for $k+1^{th}$ level and the cycle repeats till the end of k-dSLst tree.

So, **nextDiscriminator (*level_i*) = (*level_i* + 1) mod *k***.

Notations:
K: *Keys of kdSLstNode*
N: *kdSLstNode of k-dSLst tree structure*
$K_0(N), K_1(N) \ldots K_{k-1}(N)$: *k keys of kdSLstNode N*
lSon(N): *Left branch of kdSLstNode N*
rSon(N): *Right branch of kdSLstNode N*
dc(N): *Discriminator of kdSLstNode N*

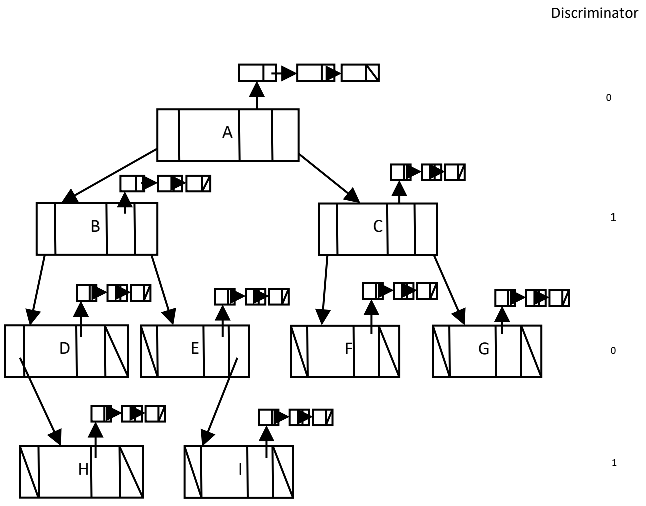


Figure 1: Structure of k-dSLst tree for 2-d keys
(It can be generalized for k-d keys)

Now, whether to insert new *kdSLstNode* as left son or right son depends on the result of comparison of keys. Let *dc* be the discriminator for *kdSLstNode N*. If $K_{dc}(N)$!= $K_{dc}(Q)$ then the successor *kdSLstNode Q* will be inserted either on left or right side of *N i.e. either lSon(N) or rSon(N)*. If $K_{dc}(N) < K_{dc}(Q)$, then *Q* will be inserted on right side of *N i.e. as rSon(N)* else on left side of *N i.e. as lSon(N)*. But, if $K_{dc}(N) = K_{dc}(Q)$, then the keys for remaining dimensions will be compared. A superkey *SK* of *kdSLstNode N* is defined by cyclical concatenation of all keys starting with $K_{dc}(N)$ as

$$SK_{dc}(N) = K_{dc}(N)\, K_{dc+1}(N) \ldots K_{dck-1}(N)\, K_{dc0}(N) \ldots K_{dc-1}(N)$$

Now if $SK_{dc}(Q) < SK_{dc}(N)$ then *Q* will be added as *lSon(N)* else *Q* will be added as *rSon(N)*.

Now, in case of duplicate keys i.e. if $SK_{dc}(Q) = SK_{dc}(N)$, then address of new *dataSNode* will be saved in the same already existing *kdSLstNode* node as a sorted linked list of *dataSNodes*.

5. Nearest Neighbour Search in k-dSLst Tree

Nearest Neighbour Search is a challenge in various domains like computer vision, spatial data mining and machine learning. There is explosive growth of location based data on the Internet and it is becoming a challenge day by day to store and manage this available data in an efficient way. The researchers have designed many indexing data structures using spatial partitions and recursive hyperplane decomposition to index spatial data. These indexing structures also speed up the nearest neighbour search. But, when it comes to spatial data with duplicate keys, many of the indexing structures do not handle them. K-dSLst tree structure is a combination of k-d tree and sorted linked list. The k-d tree having N nodes require O (Log N) inspections to search for nearest neighbour as it requires the traversal to at least one leaf of the tree. Also, as the nearest neighbour search needs to traverse a node at most once, it will not visit more than N nodes [8]. Linked list is maintained to hold all the objects at particular spatial location rather than one only.

To find nearest neighbour for any queried n-dimensional point, the search is started at the root of indexing tree and subtrees are explored recursively using the given rule of pruning spatial subtrees: If the nearest neighbour discovered up to now is nearer than the distance between the queried point and hyperplane coordinates corresponding to the current *kdSLstNode*, we can prune the exploration of this *kdSLstNode* and its subtrees further. A *kdSLstNode* needs to be explored further only if it contains point which is nearer as compared to the best one found so far. The effectiveness of the pruning rule depends on finding a nearby point. To do this, we need to organize the recursive method in a way that if we have two probable subtrees to traverse down further, we must opt for the subtree on the same side of the splitting line as is the queried point. The nearest point searched while exploring the first subtree may enable pruning of second subtree.

In this paper, we are proposing Nearest Neighbour Search **algorithm k-dSLstTreeNearest** for kdSLst tree which was designed and implemented for indexing spatial data with duplicate spatial keys. Algorithms 2 and Algorithm 3 show the proposed work. The **Algorithm 2 k-dSLstTreeNearest** takes two parameters as arguments. First is the root node of the k-dSLst tree and second is the position containing N-dimensional coordinates about which the nearest neighbour is to be searched. Rather than returning a single nearest neighbour object, the algorithm returns the list of all nearest neighbours. If root node of the tree or hyper rectangle points to NULL then INFINITY is returned back to show that there is no nearest neighbour, otherwise root node is considered as nearest neighbour to start with. The variable *distanceSqr* holds the distance in between nearest point up to now and position of point about which nearest neighbour search is queried.

Algorithm 2 k-dSLstTreeNearest

```
1:    Begin
2:    if k-dSLstTreeRoot is NULL
3:        return NIL;
4:    if k-dSLstTreeHRect is NULL
```

```
5:      return NIL;
6:   Initialize resultOfNearest_kdSLst with k-dSLstTreeRoot.
7:   Initialize distanceSqr with 0.
8:   Find distanceSqr between resultOfNearest_kdSLst and
     queryPosition.
9:   CALL k-dSLstTreeNearestIterative(k-dSLstTreeRoot,
     queryPosition,   &resultOfNearest_kdSLst, &distanceSqr, k-
     dSLstTreeHRect).
10:  if (resultOfNearest_kdSLst)
11:  then
12:     Traverse the sorted linked list of dataSNodes to show all
       objects at resultOfNearest_kdSLst
13:     Visualize resultOfNearest_kdSLst
14:  end if
15:  End
```

An iterative **Algorithm 3 k-dSLstTreeNearestIterative** is called with parameters which include root node of the k-dSLst tree, position queried about for nearest neighbour, pointer to save result node, minimum distance up to now and hyper rectangle of the tree.

Algorithm 3 k-dSLstTreeNearestIterative

```
1:   Begin
2:   Initialize currentDim to kdsLstNode's dimension
3:   Set decideLeftRight by difference in queryPosition and kdSLstNode
     coordinate values for currentDim
4:   if decideLeftRight <= 0
5:   then
6:      Set nearerSubTree to lSon(kdSLstNode)
7:      Set fartherSubTree to rSon(kdSLstNode)
8:      Update nearerHyperRectCoordinates and
        fartherHyperRectCoordinates
9:      using k-dSLstTreeHRect
10:  else
11:     Set nearerSubTree to rSon(kdSLstNode)
12:     Set fartherSubTree to lSon(kdSLstNode)
13:     Update nearerHyperRectCoordinates and
        fartherHyperRectCoordinates
14:     using k-dSLstTreeHRect
15:  end if
16:  if nearerSubTree exists
17:  then
18:     Save nearerHyperRectCoordinates to decideLeftRight
        temporarily
19:     Update nearerHyperRectCoordinates with kdSLstNode
        coordinates for  currentDim
20:     CALL k-dSLstTreeNearestIterative (nearerSubTree,
         queryPositions, resultOfNearest_kdSLst,
         resultDistanceSqr, k-dSLstTreeHRect);
21:     Update nearerHyperRectCoordinates with
22:      decideLeftRight;
23:  end if
24:  Reset distanceSqr with 0
25:  loop for every dimension dim
26:     Compute distanceSqr between kdSLstNode coordinates
        [dim] -queryPosition[dim]
27:  end loop
28:  if distanceSqr less than resultDistanceSqr
29:  then
30:     Update resultOfNearest_kdSLst and resultDistSqr
         accordingly
31:  end if
32:  if fartherSubTree exist
33:  then
34:     Save fartherHyperRectCoordinates to decideLeftRight
        temporarily
35:     Update fartherHyperRectCoordinates with kdSLstNode
         coordinates for  currentDim
36:  if closest point of k-dSLstTreeHRect is closer than
37:  resultDistanceSqr
38:  then
39:      CALL k-dSLstTreeNearestIterative (fartherSubTree,
         queryPositions, resultOfNearest_kdSLst,
        resultDistanceSqr, k-dSLstTreeHRect);
40:  end if
41:  Update fartherHyperRectCoordinates with decideLeftRight;
42:  end if
43:  end
```

Depending on the distance of query point from current point for current dimension, we decide nearer and farther sub trees and update the coordinates of nearer and farther hyper rectangles accordingly. If we find any sub tree which is nearer, we call

Algorithm 3 k-dSLstTreeNearestIterative iteratively with updated parameters. Now, check the distance of the queried point from the current point so far and update the value of *resultDistSqr*, if it is nearer. Now, also repeat the search for farther sub tree to test for any other nearer point on other side of the slice. The algorithm saves the nearest neighbour N-dimensional coordinates in *resultOfNearest_kdSLst*. At last, we traverse the complete linked list of *dataSNode* at *resultOfNearest_kdSLst* to show all of the nearest neighbours. Also, all of the nearest neighbours are displayed graphically using QGIS software.

6. Performance Evaluation: bruteForceNearestNeighbourSearch vs k-dSLstNearestNeighbourSearch

We have implemented the algorithms using **language C** and **GNU Compiler Collection (GCC) compiler - version 6.4.3** on **Operating System Ubuntu-10.04.1-Desktop-amd64**. For visualization of spatial datasets and output of **algorithms bruteForceNearestNeighbourSearch** and **k-dSLstNearestNeighbourSearch**, we have used **Quantum Geographic Information System (QGIS) Desktop 2.12.1**.

In our experiments, we are using synthetic datasets which hold spatial details of vehicles. The datasets include vehical-id, spatial coordinates (latitude, longitude) and other non-spatial attributes. We have taken six datasets of different sizes listed in Table 1 given next.

Table 1: Datasets with Duplicate Keys for Performance Evaluation

S. no.	Dataset	Number of records
1	Dataset-01	215
2	Dataset-02	430
3	Dataset-03	860
4	Dataset-04	1720
5	Dataset-05	3440
6	Dataset-06	10320

The six datasets are visualized in Figure 2 through Figure 7 using QGIS Desktop 2.12.1. These figures show the spatial location of different vehicles.

While evaluating performance of searching for nearest neighbour using **Algorithm 1 bruteForceNearestNeighbourSearch** and **Algorithm 2 k-dSLstNearestNeighbourSearch**, we are using the query point Q (43, 20), with Latitude as 43 and Longitude as 20. Both algorithms give the same results i.e. show the same Vehicles' Id as nearest neighbours for query point Q. But, the time taken by both algorithms is different. As the number of records in datasets and number of nearest neighbours increase, the algorithm **k-dSLstNearestNeighbourSearch** outperforms **bruteForceNearestNeighbourSearch**.

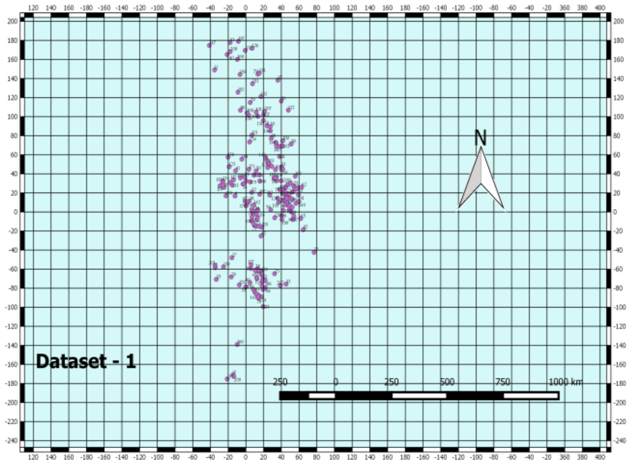


Figure 2: Vehicles' location as per Spatial Dataset – 1 (215 Spatial Records).

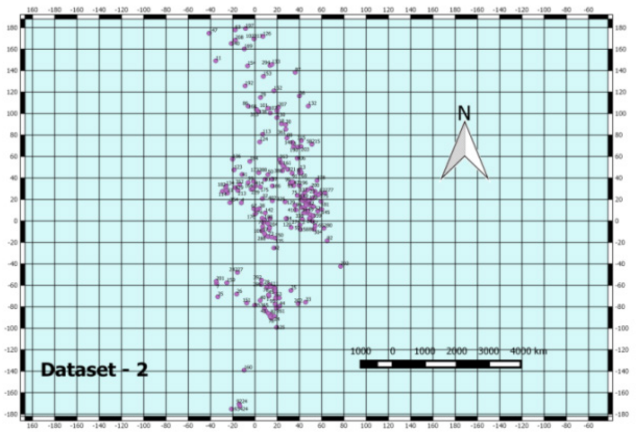


Figure 3: Vehicles' location as per Spatial Dataset – 2 (430 Spatial Records).

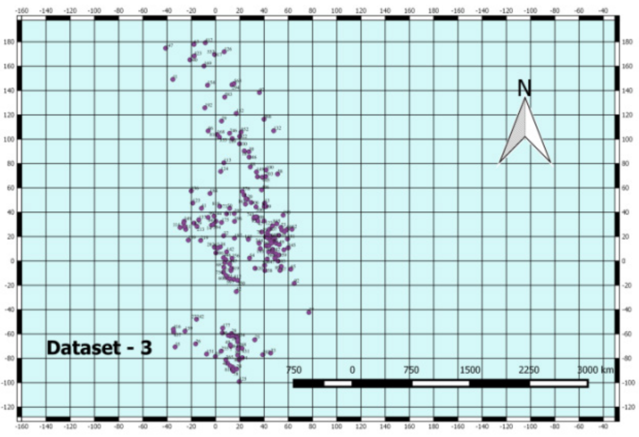


Figure 4: Vehicles' location as per Spatial Dataset – 3 (860 Spatial Records).

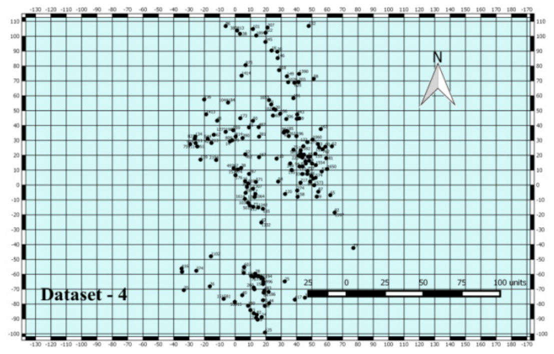


Figure 5: Vehicles' location as per Spatial Dataset – 4 (1720 Spatial Records).

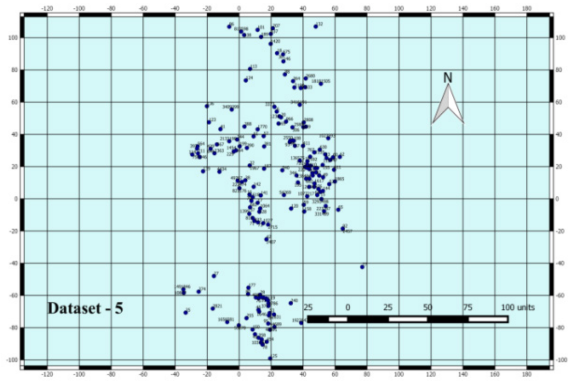


Figure 6: Vehicles' location as per Spatial Dataset – 5 (3440 Spatial Records).

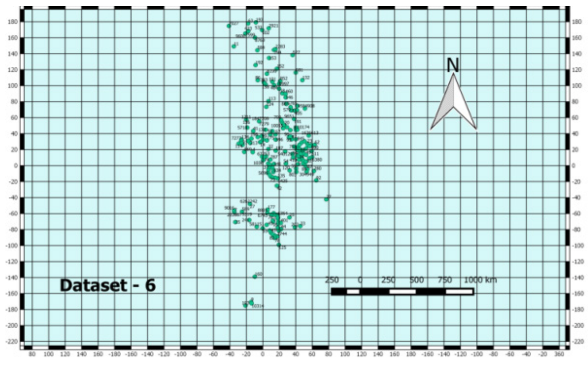
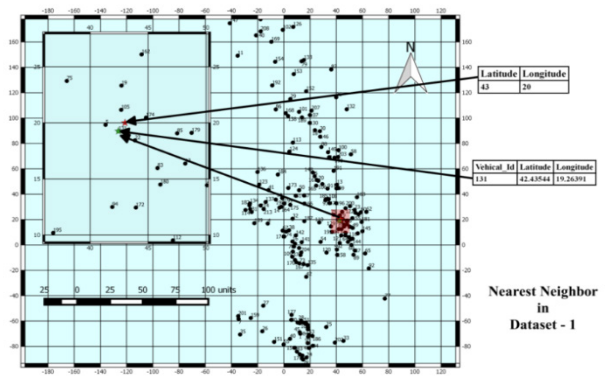


Figure 7: Vehicles' location as per Spatial Dataset – 6 (10320 Spatial Records).

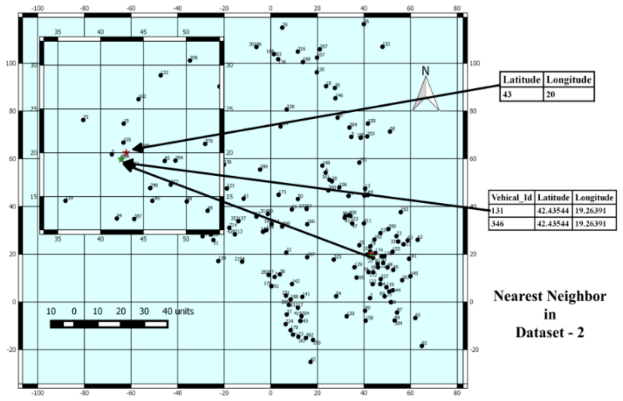


Latitude	Longitude
43	20

Vehical_Id	Latitude	Longitude
131	42.43544	19.26391

Figure 8: Nearest Neighbors of Q(43, 20) in Dataset – 1.

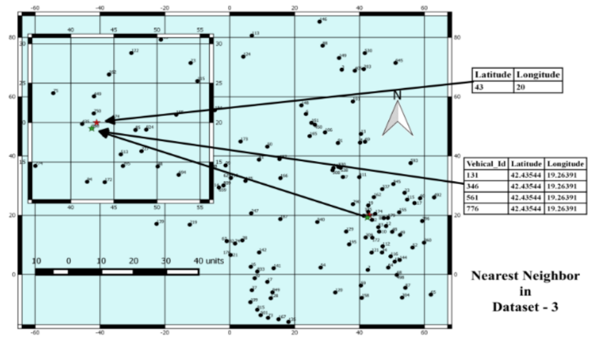
Figures 8 through 13 visualize the output of nearest neighbour query for query point Q (43, 20). Figures list Ids of all Vehicles found nearest to Q. Figures show that the algorithm **k-dSLstNearestNeighbourSearch** is well efficient to find multiple Vehicle Ids at the same nearest spatial location.



Latitude	Longitude
43	20

Vehical_Id	Latitude	Longitude
131	42.43544	19.26391
346	42.43544	19.26391

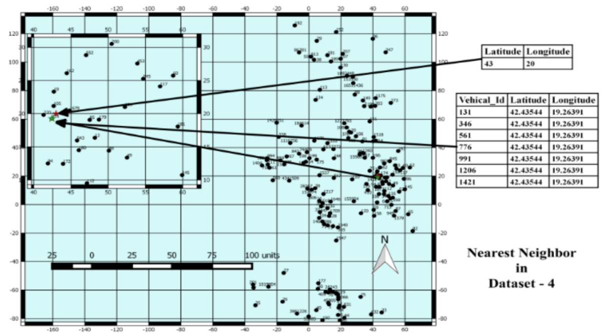
Figure 9: Nearest Neighbors of Q(43, 20) in Dataset – 2.



Latitude	Longitude
43	20

Vehical_Id	Latitude	Longitude
131	42.43544	19.26391
346	42.43544	19.26391
561	42.43544	19.26391
776	42.43544	19.26391

Figure 10: Nearest Neighbors of Q(43, 20) in Dataset – 3.

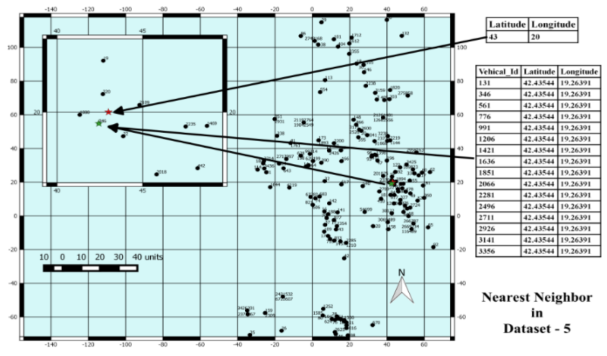


Latitude	Longitude
43	20

Vehical_Id	Latitude	Longitude
131	42.43544	19.26391
346	42.43544	19.26391
561	42.43544	19.26391
776	42.43544	19.26391
991	42.43544	19.26391
1206	42.43544	19.26391
1421	42.43544	19.26391

Figure 11: Nearest Neighbors of Q(43, 20) in Dataset – 4.

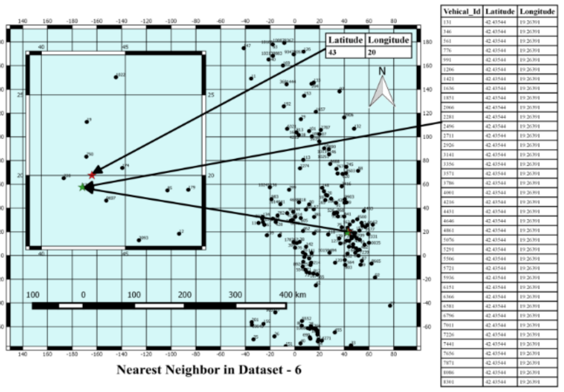
Table 2 shows the number of records found as nearest neighbours and the time taken by both algorithms in microseconds for searching the same. Results show that algorithm **k-dSLstNearestNeighbourSearch** is more efficient as compared to **bruteForceNearestNeighbourSearch** as the number of spatial records increases and capable of finding all nearest neighbours for spatial datasets with duplicate spatial keys. The search time take by algorithm **bruteForceNearestNeighborSearch** is increasing continuously and there is exponential rise from Dataset-05 to Dataset-06 as number of records increase from 3440 to 10320. Search time of algorithms is also affected by the number of records found at nearest location which are 64 in case of Dataset-05 and 192 in case of Dataset-06. As compared to algorithm **bruteForceNearestNeighborSearch** which takes 407 micro secs. and 1391 micro secs. for nearest neighbor search in Dataset-05 and Datset-06 respectively**,** algorithm **k-dSLstNearestNeighbourSearch** takes much less time i.e. 162 micro secs. and 223 micro secs. for corresponding datasets.



Latitude	Longitude
43	20

Vehical_Id	Latitude	Longitude
131	42.43544	19.26391
346	42.43544	19.26391
561	42.43544	19.26391
776	42.43544	19.26391
991	42.43544	19.26391
1206	42.43544	19.26391
1421	42.43544	19.26391
1636	42.43544	19.26391
1851	42.43544	19.26391
2066	42.43544	19.26391
2281	42.43544	19.26391
2496	42.43544	19.26391
2711	42.43544	19.26391
2926	42.43544	19.26391
3141	42.43544	19.26391
3356	42.43544	19.26391

Figure 12: Nearest Neighbors of Q(43, 20) in Dataset – 5.



Latitude	Longitude
43	20

Vehical_Id	Latitude	Longitude
131	42.43544	19.26391
346	42.43544	19.26391
561	42.43544	19.26391
776	42.43544	19.26391
991	42.43544	19.26391
1206	42.43544	19.26391
1421	42.43544	19.26391
1636	42.43544	19.26391
1851	42.43544	19.26391
2066	42.43544	19.26391
2281	42.43544	19.26391
2496	42.43544	19.26391
2711	42.43544	19.26391
2926	42.43544	19.26391
3141	42.43544	19.26391
3356	42.43544	19.26391
3571	42.43544	19.26391
3786	42.43544	19.26391
4001	42.43544	19.26391
4216	42.43544	19.26391
4431	42.43544	19.26391
4646	42.43544	19.26391
4861	42.43544	19.26391
5076	42.43544	19.26391
5291	42.43544	19.26391
5506	42.43544	19.26391
5721	42.43544	19.26391
5936	42.43544	19.26391
6151	42.43544	19.26391
6366	42.43544	19.26391
6581	42.43544	19.26391
6796	42.43544	19.26391
7011	42.43544	19.26391
7226	42.43544	19.26391
7441	42.43544	19.26391
7656	42.43544	19.26391
7871	42.43544	19.26391
8086	42.43544	19.26391
8301	42.43544	19.26391

Figure 13: Nearest Neighbors of Q(43, 20) in Dataset – 6.

Table 2: bruteForceNearestNeighbourSearch vs k-dSLstNearestNeighbourSearch

S. no.	Dataset	No. of records found at Nearest location	Time taken for search (in microseconds)	
			bruteForceNearest NeighborSearch Algorithm	k-dSLstNearest NeighbourSearch Algorithm
1	Dataset-01	4	29	106
2	Dataset-02	8	46	89
3	Dataset-03	16	90	96
4	Dataset-04	32	212	110
5	Dataset-05	64	407	162
6	Dataset-06	192	1391	223

Figure 14 shows the results of **bruteForceNearestNeighbourSearch vs k-dSLstNearestNeighbourSearch graphically.**

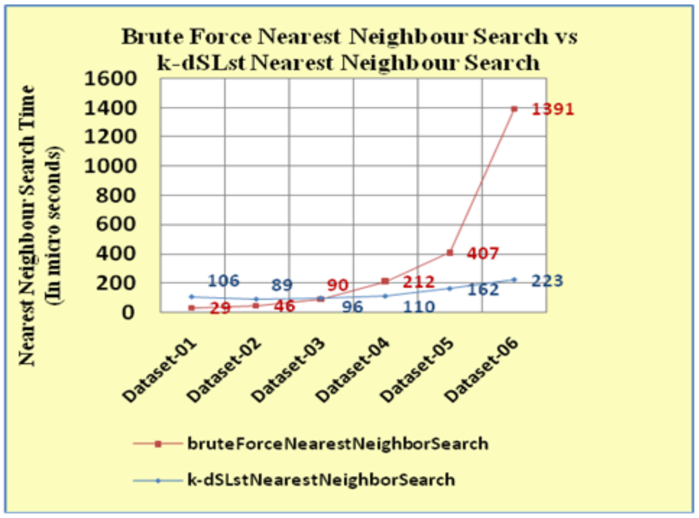


Figure 14: Search time: bruteForceNearestNeighbourSearch vs k-dSLstNearestNeighbourSearch

7. Conclusion and Future Scope

The **k-dSLstNearestNeighbourSearch algorithm** for searching nearest neighbour in spatial datasets with duplicate keys is based on **k-dSLst indexing tree structure** which is further based on k-d tree and sorted single linked list data structures. We have compared the proposed algorithm with brute-force approach. The performance evaluation shows that the search time taken by algorithm **bruteForceNearestNeighborSearch** increases exponentially as numbers on data records and nearest neighbours increases. But in case of algorithm **k-dSLstNearestNeighbourSearch** for the same datasets, the time take for search is much less and doesn't rise exponentially. The performance evaluation of both algorithms shows that the proposed work in this paper revealed better performance when compared to the conventional brute-force approach.

The work can be further expanded to search for N nearest neighbours and objects within a given range. Also, the spatial data structure k-dSLst tree can be extended to accommodate temporal data also to index spatio-temporal datasets.

References

[1] S. Shekhar et al., Spatiotemporal Data Mining: A Computational Perspective, International Journal of Geo-Information ISSN 2220-9964, 04, 2306-2338, 2015.
[2] S. Geetha, S. Velavan, "Optimization of Location Based Queries using Spatial Indexing, International Journal of Soft Computing, 4, 2014.
[3] Verma et al., "Comparison of Brute-Force and K-D Tree Algorithm, International Journal of Advanced Research in Computer and Communication Engineering, January, 03, 2014.
[4] A. Suhaibaha, et al., "3D Nearest Neighbour Search Using a Clustered Hierarchical Tree, The International Archives of the Photogrammetry, Remote Sensing and Spatial Information Sciences, XXIII ISPRS Congress, XLI-B2, 2016.
[5] J. Jaemin, et al., "A progressive k-d tree for approximate k-nearest neighbors, IEEE Workshop on Data Systems for Interactive Analysis (DSIA), 2017.
[6] W. David, R. Rafael, "Parallel kd-Tree Construction on the GPU with an Adaptive Split and Sort Strategy, International Journal of Parallel Programming, 46, 1139–1156, 2018.
[7] Stoimen. Computer Algorithms: Brute Force String Matching. Stoemen's web log. [Online] March 2012.
[8] M.W. Andrew, "Efficient Memory-based Learning for Robot Control. Technical Report No. 209, Ph. D Thesis, Computer Laboratory, University of Cambridge, 1991.