J. Cent. South Univ. (2015) 22: 4246−4253
DOI: 10.1007/s11771-015-2973-0

Springer

Approximate aggregate nearest neighbor search on moving objects trajectories

Mohammad Reza Abbasifard, Hassan Naderi, Zohreh Fallahnejad, Omid Isfahani Alamdari

MODB Lab., School of Computer Engineering, Iran University of Science and Technology, Tehran 1684613114, Iran

Abstract: Aggregate nearest neighbor (ANN) search retrieves for two spatial datasets T and Q, segment(s) of one or more trajectories from the set T having minimum aggregate distance to points in Q. When interacting with large amounts of trajectories, this process would be very time-consuming due to consecutive page loads. An approximate method for finding segments with minimum aggregate distance is proposed which can improve the response time. In order to index large volumes of trajectories, scalable and efficient trajectory index (SETI) structure is used. But some refinements are provided to temporal index of SETI to improve the performance of proposed method. The experiments were performed with different number of query points and percentages of dataset. It is shown that proposed method besides having an acceptable precision, can reduce the computation time significantly. It is also shown that the main fraction of search time among load time, ANN and computing convex and centroid, is related to ANN.

Key words: Approximate aggregate k nearest neighbor (AAkNN); scalable and efficient trajectory index (SETI); trajectory indexing; moving objects; query processing

1 Introduction

Nowadays, mobility is one of the key concepts in the society. Ubiquitous computing and location-based service (LBS) tools support mobility pretty well. The advances in location positioning and wireless communication technologies have given rise to the prevalence of mobile computing systems and location-based services, leading to a myriad of spatial trajectories representing the mobility of a variety of moving objects, such as people, vehicles, animals, and natural phenomena, in both indoor and outdoor environments [1]. Because of rapid growth in wireless communication technologies and mobile devices like smart phones, personal digital assistances (PDAs), navigational systems on vehicles such as GPS, radio-frequency identification (RFID) tags on cargos and other positioning devices, we encounter large volumes of requests for tracking moving objects in location-based systems. By using these technologies, the position of moving objects in geographical spaces can be collected. Because of production of huge volumes of data in this space (also ever increasing), it is not possible to store or process them in memory. So, appropriate methods are required for management and processing on disk.

A spatial trajectory is a trace generated by a moving object in geographical spaces. A trajectory can be modeled using a sequence of points in a three-dimensional space such that two dimensions are for spatial coordinates (x, y) and the third dimension is used for time (t) [1−3] . These three components specify the latitude and longitude of location of object and the time of presence of object at that location. By connecting these consecutive time-stamped locations, a sequence of line segments or briefly, segments, is created. The set of segments constitute a trajectory for that object. Hence, a segment of a trajectory in a k-dimensional space, in essence, is a line in (k+1)-dimensional space with time as an additional dimension [4].

Currently, moving objects trajectory data are used to response various queries like time intervals, time slice and range queries. Another important query type is k nearest neighbor (kNN) search which aims at returning k nearest neighbors to a query point. Aggregate nearest neighbor (ANN) search is a special case of NN, in which the goal is to retrieve point(s) or segment(s) with minimum aggregate distance to query points while the number of query points is high. But for various reasons such as large amount of data and continuous updates, answering these queries is very expensive by considering whether a single or a group of query points. Therefore, approximate methods are very convenient for answering these queries because they provide approximate response with an acceptable precision in a short time.

In this work, an approximate method called approximate aggregate nearest neighbor (AANN) for answering ANN queries is proposed. Normally, to answer such queries, all data must be loaded into memory. But due to large volumes of data, it would be

Received date: 2015−01−04; **Accepted date:** 2015−04−11
Corresponding author: Hassan Naderi, Assistant Professor, PhD; Tel: +98−21−73225353; E-mail: naderi@iust.ac.ir; abbasifard@iust.ac.ir

very costly. In the proposed method, firstly we compute the convex hull of query points and then, centroid of this convex hull is determined. After that, by exploiting the *k*NN algorithm, nearest segments to this centroid are obtained. These segments are close to result set of ANN with a good approximation. In experiments, in addition to evaluating the performance of our method, we have demonstrated that the approximate method has a good precision. Furthermore, currently numerous methods for management and indexing of large trajectory datasets are suggested, each of which aims at improving performance and processing of queries and ultimately reducing costs. The base for the majority of these methods is R-tree and its family like R*-tree. Since the amount of data stored on disk is very large, in this work, we used SETI indexing structure to index trajectory data. Since, in SETI, indexes are stored on disk, we have used B+-tree instead of R*-tree. Because of special characteristics of B+-tree, we would expect better performance.

2 Related works

One of the extensions to nearest neighbor search [5−7] is increasing the number of query points. This new problem is known as aggregate or group nearest neighbor problem [8−10]. In this new space, instead of one point, we have multiple query points such that our goal is to find a point from dataset of points which is near to multiple query points simultaneously. For this purpose, in these approaches, the distance relation varies and aggregate distance is proposed. The most important characteristic of such problems that are also considered in their distance relation is the simultaneous closeness of answer point to query points. The most well-known application of this approach is in "Meet Point" problem. In this problem [9], the point with minimum distance to all group members should be found. Another related topic is AggregateRoadNetwork-NNS [10]. Generally, the nearest neighbor search is introduced in Euclidean space. In the previous problem, the assumption is that the distance between any two points can be directly computed. In some applications, the problem space is Euclidean and aforementioned assumption does not hold. For instance, consider streets of a city and movement of vehicles on them. In such environments, although the space is Euclidean, there is no direct path between points. They are called road networks and modeled using planar graphs. In the established graph, the triangle inequality still holds. It should be remembered that in normal graphs, this inequality does not hold necessarily. The proposed aggregate distance in this approach is defined in the general form:

$$\text{aggdist}(p,Q)=f(\text{dist}(p,Q_1),\cdots,\text{dist}(p,Q_{n\_Q}))$$

The main functions that substitute f in above relation are: sum of distances, maximum distance and minimum distance. In order to speed up the response to aggregate k nearest neighbor queries, especially in trajectory datasets, a suitable indexing structure is needed. Efficient indexing structures [11] must be capable of supporting large volumes of trajectory data and numerous moving objects. Currently, a number of indexing structures are introduced [12−13] which can be classified into four major groups: 1) Indexing the past; 2) Indexing the current; 3) Indexing the future; 4) Indexing data at all points of time. The base for the majority of this methods is R-tree and its family [14−15].

We only concentrate on R-tree-like structures [15] that store historical information about moving object trajectories such as 3DR-tree [16−18], TB-tree [19], STRtree [19], and MV3R-tree [16]. In some indexes such as MR-tree [20], HR-tree [17, 21] and HR+-tree [22], the temporal dimension is distinguished from the spatial dimensions. One of structures that is suitable for large amounts of trajectories is SETI [4, 23] which stores its indexes on disk that is comfortable for our work.

3 Basic concepts

Fundamental concepts are described in this section. First, we introduce SETI indexing structure. Thereafter, data space partitioning of SETI is explained in detail. Next, different queries that can be answered using this method are presented.

3.1 SETI indexing structure

SETI [4, 23] is used for indexing trajectories in the past. Due to essential differences between the characteristics of spatial dimensions and the temporal dimension [24], the indexing of spatial and temporal dimensions is separated. While in structures like 3D R-tree, these dimensions are considered equal. By analyzing large trajectory datasets, one can realize that the boundaries for spatial dimensions change very slowly in the lifetime of the trajectory data, whereas the values of time dimension increase.

Because the spatial dimensions have a few change, this method logically partitions the space into static, non-overlapping cells. In the next section, data space partitioning is described. Each cell only contains those trajectory segments that are completely inside that cell. If a trajectory segment crosses the border of two cells, then it is divided into two sub-segments in the border of cells and those sub-segments are inserted into corresponding cells. Each segment is stored as a tuple in a data page on disk. Every data page only contains trajectory segments that belong to specific spatial cell. A sparse time index is built for each cell. More precisely, for every data page,

an entry is stored in the index (instead of an entry for each trajectory segment). In order to build this temporal index, the lifetime of every data page is computed. By lifetime, we mean the minimum time interval that covers all the time intervals of trajectory segments pertaining to that data page. Thereafter, all time intervals corresponding to a cell are indexed using a 1D R*-tree. Despite of being one of the best indexing methods for trajectory data, SETI has some drawbacks [23].

3.2 Data space partitioning

One of the most popular geometric shapes used in spatial data indexing is rectangle. However there are other shapes proposed for this indexing. For instance, circles are suitable for multi-dimensional indices and searching similar data. One of the main difficulties of using circles is that the real partitioning of the data space is not possible mathematically. In other words, we have to deal with overlapping. Due to large number of intersections, these overlappings result in degradation in the query performance and increased I/O costs in query processing. Generally, regular and equal-sized partitioning approaches are used in storage and retrieval and organization of spatial data, because of their simplicity in hashing and mapping to secondary storage. They provide efficient functions for this task in such a way that they firstly split data space into regular and non-overlapping rectangles and then map each cell to a physical data page [25].

Like SETI method, we partition data space into regular grid of hexagons logically. It has been proved that it is not possible to tile the plane with regular, equal-sized disjoint convex polygons with edges of more than six [25−26]. So, there are three shapes for regular partitioning: triangle, rectangle and hexagon. Among the three, hexagon is the most complex regular polygon that can partition plane without hole or overlapping.

Using hexagons has many benefits. Each hexagon has six neighbors, sharing an edge with it. The centers of all the neighbors of hexagon are of the same distance from its center. Despite of square, hexagon has no common vertex with its neighbors. This fact alone has made regular hexagons very popular for partitioning [27]. Regular hexagonal partitioning has a uniform orientation and displays a uniform adjacency. Also, considering circular range queries, hexagon is very similar to circle in terms of shape. Thus, we encounter less orientation deviation. In a regular hexagonal partitioning, the number of cells that have intersection with the range of circular query is always less than that of rectangular or triangular partitioning. That is to say, by exploiting hexagons in partitioning, we need less I/O operations [25].

The main drawback of hexagon is that it is not possible to split or join hexagons in situations like increasing or decreasing the scale of sampling. In contrast, splitting or joining a square into new squares is straightforward. For example, we can only split the hexagon into smaller triangles instead of smaller hexagons.

3.3 Queries

Applicable queries on moving objects data can be classified into two major groups [28]: coordinate-based queries and trajectory-based queries.

Examples of coordinate-based queries are time interval queries [1] in which all trajectories that have a segment contained in the specified spatial box and temporal range are returned. Another example is time slice queries [1] that select as a result all trajectories that have a segment contained in the specified spatial box at the determined time instant. Last, but not the least, nearest neighbor queries [5−7] that return all trajectories that a segment belongs to, has minimum distance to query point(s).

The trajectory-based queries are further classified into topological queries and navigational queries. Topological queries examine the whole or part of the trajectory of an object. For example, the predicate ‘collision’ examines two trajectories if they intersect in a given area at a specific time instant. The navigational query involves the information derived from the trajectories, such as the speed and the heading of an object. The average or top speed of an object is obtained by the fraction of travelled distance over time. The heading or the direction of travel of the object is computed by determining the vector between two specified positions [28].

4 Our proposed method

In our proposed method, because the amount of data is large and answering to queries is time-consuming, we use an approximate method for answering ANN queries. Figure 1 illustrates an overview of the scheme of proposed method. At first, the convex hull of query points is computed and then the centroid of this convex

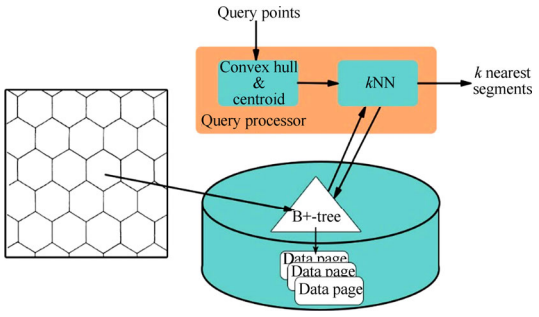

Fig. 1 Overall scheme of proposed method

is determined. After that, by exploiting the *k*NN algorithm, *k* nearest segments to this centroid are obtained. For this purpose, segments indexed by SETI structure are retrieved from disk. In order to improve the performance, we have used B+-tree instead of R*-tree as a temporal index. In the following, the proposed method is demonstrated.

4.1 Building temporal index for each cell

After partitioning data space, a temporal index for each cell is created. This procedure is useful for analyzing recursive space partitioning in trajectory indexing. 3D R-trees are widely used in indexing spatial data. A 3D R-tree is a simple method for indexing trajectories. This method extends an R-tree in a manner that can be used in a three-dimensional space such that time is considered as an additional space dimension (two dimension for space and one dimension for time). 3D R-trees are designed for coordinate-based queries like time interval and time slice queries and they are not well-suited for trajectory-based queries. There is no difference between time dimension and space dimensions and they are treated in the same manner.

In SETI method, R*-tree is used for indexing temporal dimension. In this case, for each data page, a record formatted as (data page lifetime lower bound, data page lifetime upper bound) is stored. Thus, in fact, two-dimensional data are stored in tree. Instead of R*-tree, the B+-tree is used in the proposed method. In B+-tree, only the lower bound of lifetime for each data page is stored. Since the data stored in a B+-tree are sorted, the lack of upper bound of lifetime for data pages does not affect the search process and we do not face with problematic situations throughout answering queries.

Our proposed method has some advantages:

(1) We build our index on one-dimensional data instead of two-dimensional data. Thus, the problems of dimensionality are prevented.

(2) In a B+-tree, the search time is considerably less than that of other types of trees. So, exploiting B+-tree in place of R*-tree would be beneficial.

(3) While the data on B+-tree are sorted, during range queries, the lifetime for most data pages found is in temporal range of query and there is no need for temporal propositions to be carried out on their segments in subsequent stage of query processing.

(4) Similar to SETI method, these indices are trees that are stored on disk. That is to say, when inserting the first segment in a cell, the tree corresponding to this cell is created and stored on disk. Afterward, for subsequent insert or search operations on this cell, the tree is fetched from disk into memory and upon the operation is done, the tree is stored on disk again.

Since insert operations within each cell is numerous, by using a caching method, they can be speeded up. After building the tree or after fetching it from disk, while the successive insert operations must be carried out on that cell, we can store the tree in cache memory. We can also use a mechanism like bulk-loading that is straightforward for B+-trees.

4.2 Approximate aggregate *k* nearest neighbor search

Despite of usual A*k*NN, in approximate aggregate *k* nearest neighbor search (AA*k*NN), the answer is approximate. The main goal of this method is improving the speed of search and answer to queries. In Section 4, we have shown that the approximate answer is close to main answer with a high precision. Compared with the saved time, slight loss of precision is negligible.

The best cell for finding close trajectory segments to set of query points is the cell in which the centroid of query points is located. Here, we can calculate the average of query points, and determine the center of mass by this way with $O(n)$ time complexity. This method is suitable when the goal is to minimize the sum of distances. In our proposed method, we aim at minimizing the maximum of distances. So firstly, we compute convex hull of query points and then determine the centroid of that convex hull. In addition, because the number of query points is not expected to be high, the time for computing the centroid does not matter accordingly. On the other hand, using the following way to calculate the centroid makes the cell selection process easier. For this purpose, by exploiting the Graham's scan algorithm, the convex hull of the set of query points is computed. The convex hull of a set of points in the Euclidean space is the smallest convex polygon containing all the points in that set. Graham's scan is a method of computing the convex hull of a finite set of points in the plane with time complexity of $O(n\log n)$. The algorithm finds all vertices of the convex hull ordered along its boundary. After computing the convex hull, for the resulting polygon, the centroid is determined. The way we compute the centroid is as follows. Consider a polygon made up of line segments between n vertices (x_i, y_i), i=0 to n−1. Vertices of the polygon are sequentially numbered counterclockwise from an arbitrary starting vertex. The last vertex (x_n, y_n) is assumed to be the same as the first, i.e. the polygon is closed. Then, we compute the values for S, C_x and C_y using the following relations [29]:

$$S=\frac{1}{2}\sum_{i=0}^{n-1}(x_i y_{i+1}-x_{i+1}y_i)$$

$$C_x=\frac{1}{6A}\sum_{i=0}^{n-1}(x_i+x_{i+1})(x_i y_{i+1}-x_{i+1}y_i)$$

$$C_y = \frac{1}{6A}\sum_{i=0}^{n-1}(y_i + y_{i+1})(x_i y_{i+1} - x_{i+1} y_i)$$

where S is the area of the polygonal region as calculated by the coordinate area formula and (C_x, C_y) are the coordinates of the centroid. Very general polygonal regions may be evaluated using this formula including disconnected polygons and those with holes [29]. One advantage of the preceding formula is that for a polygonal region, it is exact. After determining the centroid, the search process begins. The cell containing the centroid point is the first cell for search process to start from.

Hereafter, the search process would be similar to *k*NN search. All the segments in data pages belonging to this cell are investigated and the set of their distances to query points is computed. In the implemented method, for each segment, the sum of all distances to query points is determined. Then, for each trajectory, the distance that is minimum among its belonging segments is assigned (The distance values for segments of that trajectory are compared and the smallest is assigned to it). Ultimately, k first trajectories close to query point are returned as answer (These are trajectories that have a segment with minimum sum of distances). The distance between a query point q and trajectory A is usually measured by the distance from the point q to the nearest point on trajectory A [1, 30]:

$$\text{dist}(q, A) = \min_{p' \in A}(q, p')$$

So, the distance of a segment s to set of query points Q is computed as follows:

$$\text{dist}(s, Q) = \text{sum}(\text{dist}(s, Q_1), \cdots, \text{dist}(s, Q_{n\_Q}))$$

And the distance of a trajectory A to set of query points Q can be obtained as

$$\text{dist}(A, Q) = \min_{s' \in A}[\text{dist}(s', Q)]$$

By extending one query point to multiple points, this relation can be used:

$$\text{sim}(Q, A) = \sum_{q \in Q} \exp[\text{dist}(q, A)]$$

The intuition of using the exponential function is to assign a larger contribution to pairs of points that are close to each other and a small contribution to pairs that are far away. As a result, only trajectories that their points are closer in all query locations are selected as similar trajectories.

5 Performance experiments

For implementation of our experiments, we used a system with Core i5 3.30 GHz Intel CPU and 8 GB main memory, running Windows 8. Also, we implemented algorithms in Java programming language. We used trajectory dataset of Microsoft Geolife project for evaluation. This GPS trajectory dataset is collected by 182 users in a period of over five years. A GPS trajectory in this dataset is represented as a sequence of points, each of which consists of information about latitude and longitude, altitude and time and date of sampling. In order to remove noisy and outlier data, firstly, we did a preprocessing on data.

Next, to evaluate the effect of dataset size on speed and precision of the proposed method, the improved index was built for the certain percentages of dataset, namely 10%, 20%, 30%, 40%, 50% and 100%. Then, on each of the resulting set of indexes, different search methods like time interval, time slice, aggregate k nearest neighbor (A*k*NN) and approximate A*k*NN (AA*k*NN) were experimented. However, AA*k*NN is more important than other search methods and we concentrated on that. For this purpose, we selected some random points from intended spatial range as query points. For better evaluation of the obtained results, the number of query points increased gradually such that 1, 10, 100, 500, 1000, 2000, 5000, 10000, 20000, 50000 and 100000 query points were considered. That is to say, in the case of one query point, only the *k*NN algorithm and for higher number of query points, A*k*NN and AA*k*NN algorithms were carried out. In the next section, we present results of experiments on speed and precision of aforementioned algorithms.

5.1 Evaluation results

Due to huge volumes of dataset stored on disk, answering ANN queries needs an access to disk for each query point. So, this I/O operation would be very time-consuming. But, in approximate method proposed in this work, the query points are reduced to one point which is the centroid of them. This leads to a significant reduction in disk accesses.

As mentioned above, we did experiments on different index percentages and different number of query points. For each experiment, in every stage, the time for calculating convex hull and centroid, and the load time and A*k*NN and AA*k*NN times are obtained. In the end, the runtime of each experiment is calculated. The results of experiments is shown for different index sizes and plotted.

Experiments show that, calculation of convex hull and centroid of query points takes a negligible time of the whole search process. As indicated in Fig. 2, if the number of query points is large (e.g. larger than 10000), the whole search process is slightly affected by the convex hull and centroid calculation time. But, it should be said that, usually we never encounter with such a high number of query points in practice. Meanwhile,

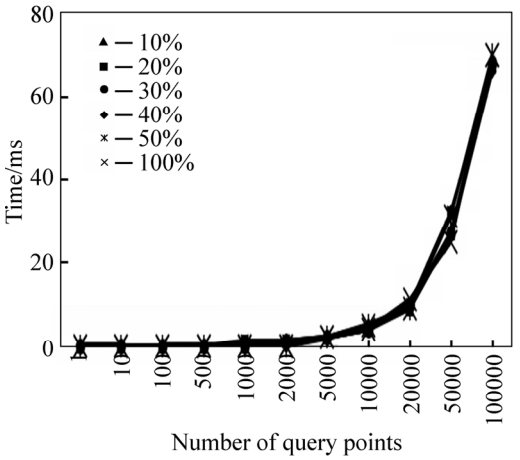

Fig. 2 Convex hull and centroid calculation time

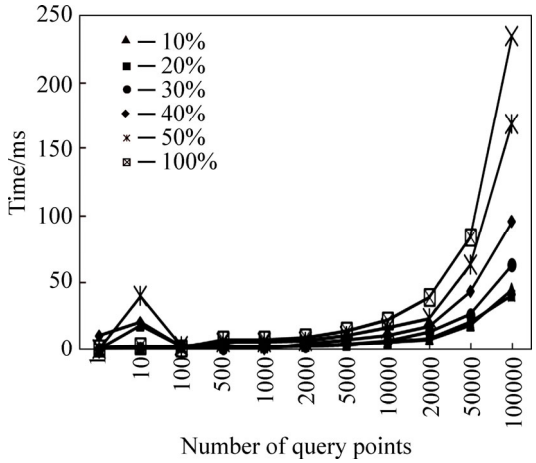

Fig. 4 Approximate aggregate *k*NN time

increasing the size of dataset does not affect the convex hull and centroid calculation time.

Another case studied in experiments is load time of data pages and sparse temporal indices from disk to main memory. This time is apart from the runtime and usually is related to operating system and hardware used. It should be noted that by increasing the size of dataset, the load time increases. But the increase in the number of query points has no effect on load time. At the same time, by exploiting different techniques like caching, the load time can be reduced. Figure 3 depicts the load time of different dataset sizes.

The main time in search process is related to AA*k*NN computation. Like calculating convex hull and centroid, this time also does not change much by increasing the number of query points unless there are too many query points. For example, as shown in Fig. 4, the AA*k*NN time matters if the number of query points exceeds 20000. However, this case is very rare in reality. Also, the increase in the size of dataset has a little effect on the AA*k*NN time. As seen in Fig. 4, if the number of query points is greater than 20000, the time for larger datasets is increased. That is to say, the load time is also considered. For the less number of query points (less than 2000), the load time takes more than half of time. But by increasing the number of query points, the effect of load time is decreased.

Another time computed in experiments is runtime that consists of convex hull and centroid calculation, AA*k*NN, sorting and representation of results. Since this time is taken from abovementioned times, it has a behavior similar to them. In Fig. 5, the results of runtime are shown.

The convex hull and centroid calculation, load and AA*k*NN time contributions in the whole search process are shown in Fig. 6. As illustrated, the main time pertains to computing AA*k*NN. If the number of query points increases (greater than 20000), then the effect of convex hull and centroid calculation becomes higher.

Another important point evaluated in experiments is assessing the precision of the proposed method (AA*k*NN) compared with common A*k*NN methods. To calculate the precision for different query points, in each experiment, the average distance for first 10 answers in both AA*k*NN and A*k*NN methods is calculated and the comparison is reported in Table 1. Furthermore, Fig. 7 depicts the

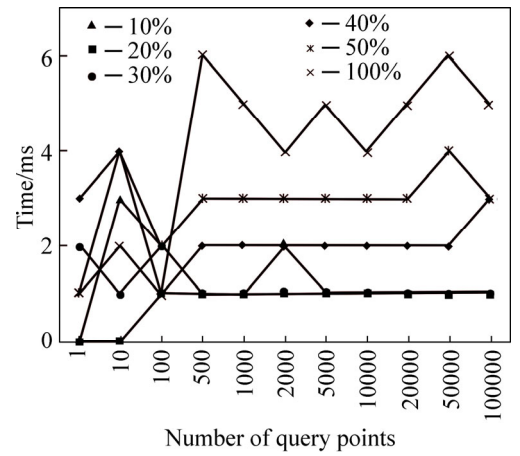

Fig. 3 Data pages and index load time

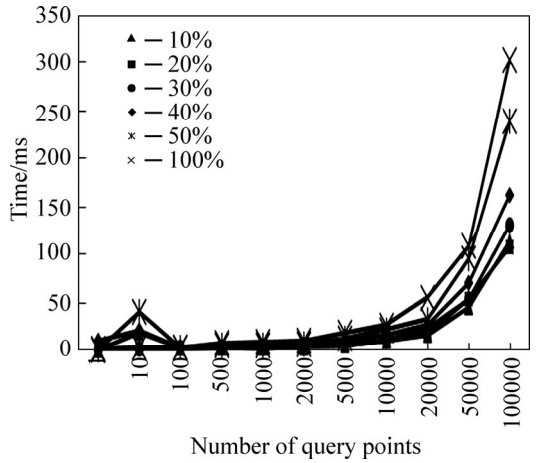

Fig. 5 Total runtime

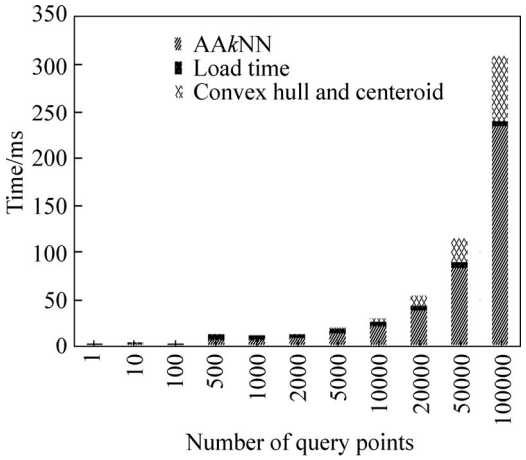

Fig. 6 Contribution of three major times in whole search process

Table 1 Comparison of average distance for query results

Number of query points	Distance	
	AA*k*NN	A*k*NN
1	0.00	0.00
10	1.37	1.31
100	15.88	15.62
1000	157.10	155.88
2000	302.66	300.13
5000	762.93	757.24

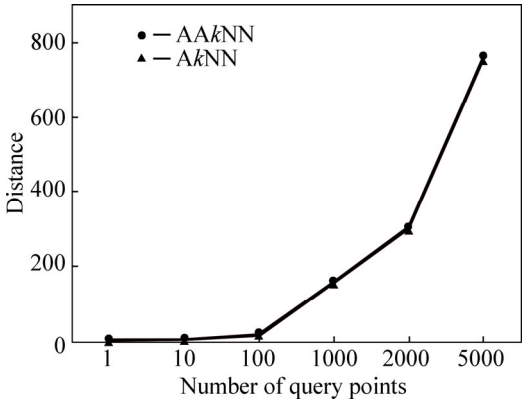

Fig. 7 Comparison of average distance for query results

precision of AA*k*NN compared to A*k*NN. Considering that the AAkNN is an approximate method, but in experiments we realized that its precision is close to common methods, it is a suitable method.

In addition, computing A*k*NN in common methods needs a vast amount of data pages and corresponding sparse time indexes to be loaded into memory. Moreover, calculating the distance of query points to segments of each cell takes too much time. Totally, common methods waste considerable time and memory to prepare answers. But, AA*k*NN method can greatly reduce the computation time with precision guaranty.

6 Conclusions and future work

1) An approximate method for retrieving one or more segments from trajectories of dataset having minimum aggregate distance to query points, was proposed.

2) Our main idea is the reducing of number of query points to one query point, consequently to improve response time of queries over large amounts of trajectories. In this idea, despite of large volume of dataset and numerous query points, accessing to disk is considerably reduced.

3) In our proposed method, the convex hull of query points is calculated and for this convex hull, centroid is determined and then segments with minimum distance to this centroid are returned as result.

4) The SETI structure is refined for indexing large amounts of trajectories in such a way that its two-dimensional temporal index is substituted by a one-dimensional index to improve the performance.

5) For future works, diverse Big Data applications can benefit from this idea. Since in Big Data applications, data are stored on disk, and by exploiting this method, accessing to disk can be reduced.

6) In order to get further improvement in time and precision, we plan to use this method in other indexing structures like TB-tree. Also, we intend to exploit this method in structures used for indexing trajectories of present and future times.

References

[1] ZHENG Yu, ZHOU Xiao-fang. Computing with spatial trajectories [M]. New York: Springer, 2011: 3−60.

[2] PARENT C, SPACCAPIETRA S, RENSO C, ANDRIENKO G, ANDRIENKO N, BOGORNY V, DAMIANI M L, GKOULALAS-DIVANIS A, MACEDO J, PELEKIS N, THEODORIDIS Y, YAN Z. Semantic trajectories modeling and analysis [J]. ACM Computing Surveys (CSUR), 2013, 45: 1−32.

[3] RENSO C, SPACCAPIETRA S, ZIMÁNYI E. Mobility data modeling management and understanding [M]. Cambridge University Press, 2013: 5−10.

[4] CHAKKA V P, EVERSPAUGH A C, PATEL J M. Indexing large trajectory data sets with SETI [C]// The Conference on Innovative Data Systems Research (CIDR 2003). USA, 2003.

[5] ABBASIFARD M R, GHAHREMANI B, NADERI H. A survey on nearest neighbor search methods [J]. International Journal of Computer Applications, 2014, 95: 39−52.

[6] DHANABAL S, CHANDRAMATHI S. A Review of various *k*-nearest neighbor query processing techniques [J]. Computer Applications, 2011, 31: 14−22.

[7] BHATIA N, ASHEV V. Survey of nearest neighbor techniques [J]. International Journal of Computer Science and Information Security, 2010, 8: 1−4.

[8] PAPADIAS D, SHEN Q, TAO Y, MOURATIDIS K. Group nearest neighbor queries [C]// 20th International Conference on Data Engineering, Massachusetts, Boston, USA, 2004: 301−312.

[9] PAPADIAS D, TAO Y, MOURATIDIS K, HUI C K. Aggregate nearest neighbor queries in spatial databases [J]. ACM Transactions on Database Systems, 2005, 30: 529−576.

[10] YIU M L, MAMOULIS N, PAPADIAS D. Aggregate nearest neighbor queries in road networks [J]. IEEE Transactions on Knowledge and Data Engineering, 2005, 17: 820−833.

[11] FENG J, WATANABE T. Index and query methods in road networks [M]. Springer International Publishing, 2015: 11−39.

[12] MOKBEL M F, GHANEM T M, AREF W G. Spatio-temporal access methods [J]. IEEE Data Engineering Bulletin, 2003, 26: 40−49.

[13] NGUYEN-DINH L, AREF W G, MOKBEL M F. Spatio-temporal access methods: Part 2 (2003-2010) [J]. IEEE Data Engineering Bulletin, 2010, 33: 46−55.

[14] GUTTMAN A. R-trees: A dynamic index structure for spatial searching [J]. ACM SIGMOD Record, 1984, 14: 47−57.

[15] MANOLOPOULOS Y, NANOPOULOS A, PAPADOPOULOS A N, THEODORIDIS Y. R-Trees: Theory and applications [M]. New York: Springer, 2005: 3−20.

[16] TAO Y, PAPADIAS D. MV3R-Tree: A spatio-temporal access method for timestamp and interval queries [C]// The 27th International Conference on Very Large Data Bases (VLDB '01). Rome, 2001: 431−440.

[17] NASCIMENTO M A, SILVA J R O, THEODORIDIS Y. Evaluation of access structures for discretely moving points [C]// The International Workshop on Spatio-Temporal Database Management (STDBM'99). Edinburgh, 1999: 171−188.

[18] THEODORIDIS Y, VAZIRGIANNIS M, SELLIS T. Spatio-temporal indexing for large multimedia applications [C]// The Third IEEE International Conference on Multimedia Computing and Systems (ICMCS '96). Hiroshima, 1996: 441−448.

[19] PFOSER D, JENSEN C S, THEODORIDIS Y. Novel Approaches in query processing for moving object trajectories [C]// 26th International Conference on Very Large Data Bases (VLDB '00). Cairo, Egypt, 2000: 395−406.

[20] XU X, HAN J, LU W. RT-tree: An improved R-tree indexing structure for temporal spatial databases [C]// The International Symposium on Spatial Data Handling (SDH). Zurich, 1990: 1040−1049.

[21] NASCIMENTO M A, SILVA J R O. Towards historical R-trees [C]// The 1998 ACM symposium on Applied Computing (SAC '98). Atlanta, 1998: 235−240.

[22] TAO Y, PAPADIAS D. Efficient historical R-trees [C]// 13th International Conference on Scientific and Statistical Database Management (SSDBM '01). Fairfax, 2001: 223.

[23] CHA Chang-il, KIM Sang-wook, WON Jung-im, LEE Junghoon, BAE Duck-ho. Efficient indexing in trajectory databases [J]. International Journal of Database Theory and Application, 2008, 1(1): 21−28.

[24] SONG R, SUN W, ZHENG B, ZHENG Y. PRESS: A novel framework of trajectory compression in road networks [C]// Proceedings of the VLDB Endowment. Hangzhou, 2014: 661−672.

[25] FERHATOSMANOĞLU H, AGRAWAL D, EĞECIOĞLU Ö, EL ABBADI A. Optimal data-space partitioning of spatial data for parallel I/O [J]. Distributed and Parallel Databases, 2005, 17(1): 75−101.

[26] GRUNBAUM B, SHEPHARD G C. Tilings by regular polygons [J]. Mathematics Magazine, 1977, 50: 227−247.

[27] SAHR K, WHITE D, KIMERLING A J. Geodesic discrete global grid systems [J]. Cartography and Geographic Information Science, 2003, 30: 121−134.

[28] ANTOINE E, RAMAMOHANARAO K, SHAO J, ZHANG R. Recursive partitioning method for trajectory indexing [C]// The Twenty-First Australasian Conference on Database Technologies-Volume 104 (ADC '10). Darlinghurst, Australia, 2010: 37−46.

[29] GREULICH F E. Accurate polygon centroid computation using ARC/INFO GIS [J]. Journal of Computing in Civil Engineering, 1993, 7: 388−392.

[30] CHEN Z, SHEN H T, ZHOU X, ZHENG Y, XIE X. Searching trajectories by locations−An efficiency study [C]// The 2010 ACM SIGMOD International Conference on Management of data (SIGMOD '10). Indianapolis, Indiana, USA, 2010: 255−266.

(Edited by YANG Bing)