CircularTrip: An Effective Algorithm for Continuous kNN Queries

Muhammad Aamir Cheema, Yidong Yuan, and Xuemin Lin

The University of New South Wales, Australia
{macheema, yyidong, lxue}@cse.unsw.edu.au

Abstract. Continuously monitoring kNN queries in a highly dynamic environment has become a necessity to many recent location-based applications. In this paper, we study the problem of continuous kNN query on the dataset with an in-memory grid index. We first present a novel data access method – CircularTrip. Then, an efficient CircularTrip-based continuous kNN algorithm is developed. Compared with the existing algorithms, our algorithm is both space and time efficient.

1 Introduction

Continuously monitoring k nearest neighbors over moving data objects has become a necessity to many recent location-based applications. This is mainly due to the increasing availability of wireless networks and inexpensive mobile devices. Consequently, a number of techniques [1,2,3,4,5,6,7,8,9] have been developed to process continuous kNN queries.

Different from a conventional kNN query, continuous kNN queries are issued once and run continuously to generate results in real-time along with the updates of the underlying datasets. Therefore, it is crucial to develop in-memory techniques to continuously process kNN queries due to frequent location updates of data points and query points. In many applications [6,7,9], it is also crucial to support the processing of a number of continuous kNN queries simultaneously; consequently, scalability is a key issue.

To address the scalability, in this paper we focus on two issues: (1) minimization of computation costs; and (2) minimization of the memory requirements. We study continuous kNN queries against the data points that move around in an arbitrary way. To effectively monitor kNN queries, we develop a novel data access method – CircularTrip. Compared with the most advanced algorithm, CPM [9], our CircularTrip-based continuous kNN algorithm has the following advantages. (1) time efficient: although both algorithms access the minimum number of cells for initial computation, less cells are accessed during continuous monitoring in our algorithm. (2) space efficient: our algorithm does not employ any book-keeping information used in CPM (i.e., visit list and search heap for each query). Our experimental study demonstrates that CircularTrip-based continuous kNN algorithm is 2 to 4 times faster than CPM, while its memory usage is only 50% to 85% of CPM.

R. Kotagiri et al. (Eds.): DASFAA 2007, LNCS 4443, pp. 863–869, 2007.

Our contributions in this paper can be summarized as follows: (1) We develop a novel data access method – CircularTrip which returns the cells intersecting a given circle with the minimum number of cells examined; (2) Based on CircularTrip, a time- and space- efficient continuous kNN algorithm is developed.

The rest of the paper is organized as follows: Section 2 gives the problem definition and presents the related work. We present our continuous kNN algorithm in Section 3. Experimental study and remarks are reported in Section 4.

2 Background Information

Suppose that P is a set of 2D[1] moving data points and data points change their locations frequently in an unpredictable fashion. Each data point $p \in P$ is represented by $(p.x, p.y)$. At each time stamp, the move of a data point p from p_{pre} to p_{cur} is recorded as a location update $\langle p.id, p_{pre}, p_{cur} \rangle$ and the moves of query points are recorded similarly. The problem of continuous kNN query is formally defined below.

Continuous kNN Query. *Given a set of moving data points, a moving query point q, and an integer k, the continuous kNN query is to find k closest data points to q at each time stamp.*

Grid Index. In this paper, we assume that the dataset is indexed by an in-memory grid index which evenly partitions the space into *cells*. The extent of each cell on each dimension is δ. Cell $c[i, j]$ indicates the cell at column i and row j and the lower-left corner cell is $c[0, 0]$. Clearly, point p falls into the cell $c[\lfloor p.x/\delta \rfloor, \lfloor p.y/\delta \rfloor]$.

In the grid index, each cell is associated with an *object list* and an *influence list*. Object list contains all the data points in this cell. Influence list of cell c maintains the references of all the queries q such that $mindist(c, q) \leq q.dist_k$ where $q.dist_k$ is the distance of k^{th} nearest neighbor from q. Specially, $mindist(c^q, q) = 0$ where c^q is the cell containing q. Note that both object list and influence list are implemented as hash tables so that lookup, insertion, update, and deletion of entries take constant time.

Accessing and *encountering* are two basic operations on cells. Specifically, accessing a cell is to evaluate all data points in this cells against queries and encountering a cell only computes its minimum distance to queries. Clearly, cost of encountering a cell is neglected when compared with accessing a cell.

SEA-CNN [6], YPK-CNN [7], and CPM [9] are the most related existing work to the study in this paper. Due to space limitation, we omit the details of these techniques here. Interested readers can find them in [6,7,9], respectively.

3 Continuous kNN Algorithm

Our CircularTrip-based continuous kNN algorithm consists of two phases. In phase 1, the initial results of each new continuous kNN query is computed. Then,

[1] In this paper, we focus on 2D space only. But the proposed techniques can be applied to higher dimensional space immediately.

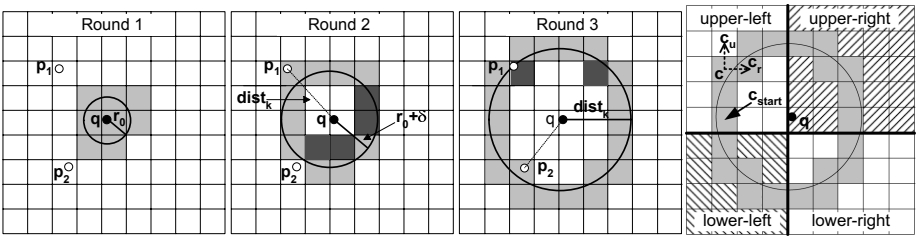


Fig. 1. An NN Query

Fig. 2. CircularTrip

the results are incrementally updated by continuous monitoring module at each time stamp upon the moves of query points and data points (i.e., phase 2). Both phases take advantages of CircularTrip algorithm. Section 3.1 and Section 3.2 present phase 1 and phase 2, respectively.

3.1 Initial kNN Computation

The basic idea of kNN computation algorithm is to access the cells around query point q round by round. A *round* C_i contains all the cells that intersect the circle of radius $r_i = r_0 + i\delta$ centered at q. Formally, $C_i = \{\forall c \mid mindist(c,q) < r_i \leq maxdist(c,q)\}$. r_0 is the the first circle's radius. Obviously, r_0 is at most $maxdist(c^q, q)$; otherwise cell c^q will not be accessed by the algorithm. Examples of round are shown as the shaded cells in Fig. 1. In each round, the algorithm accesses the cells in ascending order of their $mindist(c,q)$. The algorithm terminates when the next cell to be accessed has $mindist(c,q) \geq q.dist_k$. The following theorem proves the correctness and optimality of this algorithm.

Lemma 1. *In a grid consisting of cells with size $\delta \times \delta$, given a cell c and a query point q where c does not contain q, $\delta \leq maxdist(c,q) - mindist(c,q) \leq \sqrt{2}\delta$.* □

Theorem 1. *Given a query q, in our initial kNN algorithm, the minimal set of cells are all accessed and only these cells are accessed.* □

According to Lemma 1, a cell is intersected by at most two consecutive circles (e.g., the dark shaded cells in Fig. 1). Although these cells are encountered twice during kNN computation (i.e., these cells appear in two rounds), they are accessed once only. This is because for a query q (1) our kNN algorithm only accesses the cells where q is not in their influence lists; and (2) q will be inserted into its influence list after a cell is processed. In fact, Theorem 2 proves the upper bound of the total number of times the cells are encountered in our algorithm.

Theorem 2. *In kNN algorithm, the total number of times the cells are encountered is at most* 1.27 *times of the number cells in the minimum set of cells.* □

The detailed kNN computation algorithm is shown in Algorithm 1. We use the following example to present its details.

Algorithm 1. ComputeNN(G, q, k)

Input: G: the grid index; q: query point; k: an integer;
Output: the kNN of q;

```
1: q.dist_k := ∞; q.kNN := ∅; H := ∅; r := r_0 := maxdist(c^q, q);
2: insert the cells returned by CircularTrip(G, q, r) into H;
3: while H ≠ ∅ and mindist(e^H, q) < q.dist_k do
4:    insert q into the influence list of e^H;
5:    ∀p ∈ e^H, compute dist(p, q) and update q.dist_k and q.kNN;
6:    remove e^H from H;
7:    if H = ∅ and r < q.dist_k then
8:       r := min{r + δ, q.dist_k};
9:       cells C := CircularTrip(G, q, r);
10:      ∀c ∈ C, insert c into H if q ∉ the influence list of c;
11: return q.kNN;
```

Example 1. Fig. 1 illustrates a concrete example of an NN query. As no data point is found in the first round, the algorithm continues to process the cells in the next round with radius $(r_0 + \delta)$. In this round, p_1 is found and $q.dist_k$ is updated to be $dist(p_1, q)$. Then, a third round with radius $q.dist_k$ (as $dist(p_1, q) < r_0 + 2\delta$) is processed because the previous radius is smaller than $q.dist_k$. In round 3, $q.kNN$ and $q.dist_k$ are updated after p_2 is found. Computation stops when $q.dist_k$ $(= dist(p_2, q))$ is less than $mindist(e^H, q)$ of the top entry e^H.

CircularTrip Algorithm. To collect a round of cells, CircularTrip starts from one cell intersected by the given circle and checks the cells along the circle. Without loss of generality, consider cell c intersected by the circle which locates in the upper-left quadrant as shown in Fig. 2. The key fact is that the next cell intersected by the circle (i.e., the cell in which the arc is connected to one in c) is the adjacent cell either above c (i.e., c_u) or right to c (i.e., c_r). This is because the outgoing circle crosses either the upper boundary or the right boundary of c. These two adjacent cells, c_u and c_r, are called *candidate adjacent cells* of c. Clearly, to collect the next cell intersected by the circle, CircularTrip only needs to examine one of the candidate adjacent cells (i.e., check its $mindist(c, q)$ with the given radius r). As a result, the total cost of CircularTrip to collect a round C of cells is to compute $mindist(c, q)$ of $|C|$ cells, where $|C|$ is the number of cells in round C.

Algorithm 2 presents the implementation of CircularTrip algorithm. It always starts from the left most cell of the round c_{start} (as shown in Fig. 2) and examines the cells clockwise along the given circle until c_{start} is encountered again. When the quadrant of the current cell being examined is changed, the directions to find its candidate adjacent cells are updated accordingly (i.e., lines 9 – 10).

3.2 Continuous Monitoring

Same as in CPM, when the query moves, we simply re-issue the query on the new location. So, continuous monitoring only concerns update of data points.

Algorithm 2. CircularTrip(G, q, r **)**

Input: G: the grid index; q: query point; r: the radius;
Output: all the cells which intersect the circle with center q and radius r;

1: $C := \varnothing$; $c := c_{start} := c[i,j]$ ($i := \lfloor (q.x - r)/\delta \rfloor$, $j := \lfloor q.y/\delta \rfloor$);
2: $D_{cur} := Up$; /* `clockwise fashion:` $Up \rightarrow Right \rightarrow Down \rightarrow Left \rightarrow Up$ */
3: **repeat**
4: insert c into C;
5: c' := the adjacent cell to c in D_{cur} direction;
6: **if** c' does not intersect the circle **then**
7: c' := the adjacent cell to c in the next direction of D_{cur};
8: $c := c'$;
9: **if** ($c.i = c^q.i$ and $c.j = \lfloor (q.y \pm r)/\delta \rfloor$) or ($c.i = \lfloor (q.x \pm r)/\delta \rfloor$ and $c.j = c^q.j$) **then**
10: D_{cur} := the next direction of D_{cur};
11: **until** $c = c_{start}$
12: **return** C;

Regarding a query q, the update of data point p, $\langle p.id, p_{pre}, p_{cur} \rangle$, can be classified into 3 cases:

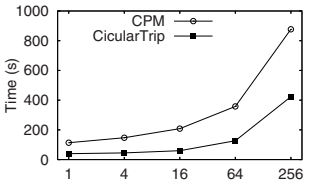
- *internal update*: $p_{cur} \in q.kNN$ and $p_{pre} \in q.kNN$; clearly, only the order of $q.kNN$ is affected so we update the order of data points in $q.kNN$ accordingly.
- *incoming update*: $p_{cur} \in q.kNN$ and $p_{pre} \notin q.kNN$; p is inserted in $q.kNN$.
- *outgoing update*: $p_{cur} \notin q.kNN$ and $p_{pre} \in q.kNN$; p is deleted from $q.kNN$.

It is immediately verified that only the queries recorded in the influence lists of cell $c^{p_{pre}}$ or cell $c^{p_{cur}}$ may be affected by the update $\langle p.id, p_{pre}, p_{cur} \rangle$, where $c^{p_{pre}}$ ($c^{p_{cur}}$) is the cell containing p_{pre} (p_{cur}). Therefore, after receiving an update $\langle p.id, p_{pre}, p_{cur} \rangle$, continuous monitoring module checks these queries q only. If $dist(p_{cur}, q) \leq q.dist_k$, it is treated as an incoming update (if $p_{pre} \notin q.kNN$) or an internal update (if $p_{pre} \in q.kNN$). On the other hand, If $dist(p_{pre}, q) \leq q.dist_k$ and $dist(p_{cur}, q) > q.dist_k$, it is handled as an outgoing update.

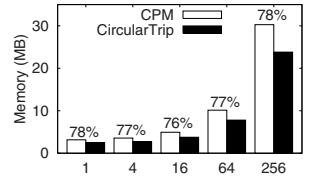
After all the updates of data points are handled as described above, we update the results of affected queries. For each query q, if $|q.kNN| \geq k$, we keep the k closest points and delete all other. For any query q where $|q.kNN| < k$, we update its result in a similar way to Algorithm 1. Note that here the starting radius r_0 is set as $q.dist_k$. The intuition is the fact that any update within this distance has already been handled.

4 Experimental Study and Remarks

In accordance with the experimental study of previous work [6,9], we use the same spatio-temporal data generator [10]. Data points with slow speed move 1/250 of the extent of space per time stamp. Medium and fast speed are 5 and 25 times faster than slow speed, respectively. Continuous kNN queries are

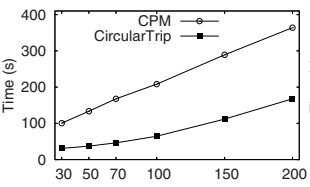


(a) Time

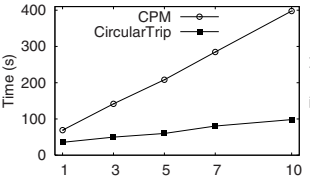


(b) Space

Fig. 3. Effect of k

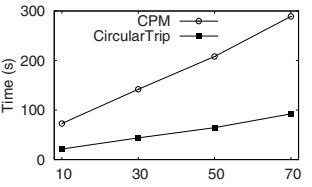


(a) Varying N ($\times 1K$)

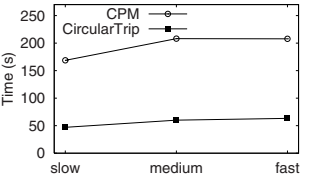


(b) Varying n ($\times 1K$)

Fig. 4. Effect of N and n



(a) Varying Agility (%)



(b) Varying Speed

Fig. 5. Data Movement

generated in the similar way. All queries are evaluated at each time stamp and the length of evaluation is 100 time stamps. The grid index has 256×256 cells.

We evaluate our CircularTrip-based continuous kNN technique against various parameters: number of NNs (k), number of data points (N), number of queries (n), and data point agility and moving speed. In our experiments, their default values are 16, $100K$, $5K$, 50%, and *medium*, respectively. The experimental results are reported in Fig. 3, 4, and 5.

In this paper, we develop an efficient CircularTrip-based continuous kNN algorithm. Compared with the existing algorithm, our technique accesses the minimum set of cells for initial computation and significantly reduces the continuous monitoring cost, while less memory space is required.

References

1. Song, Z., Roussopoulos, N.: K-nearest neighbor search for moving query point. In: SSTD. (2001) 79–96
2. Tao, Y., Papadias, D.: Time-parameterized queries in spatio-temporal databases. In: SIGMOD Conference. (2002) 334–345
3. Tao, Y., Papadias, D., Shen, Q.: Continuous nearest neighbor search. In: VLDB. (2002) 287–298
4. Zhang, J., Zhu, M., Papadias, D., Tao, Y., Lee, D.L.: Location-based spatial queries. In: SIGMOD. (2003) 443–454
5. Iwerks, G.S., Samet, H., Smith, K.P.: Continuous k-nearest neighbor queries for continuously moving points with updates. In: VLDB. (2003) 512–523
6. Xiong, X., Mokbel, M.F., Aref, W.G.: Sea-cnn: Scalable processing of continuous k-nearest neighbor queries in spatio-temporal databases. In: ICDE. (2005) 643–654
7. Yu, X., Pu, K.Q., Koudas, N.: Monitoring k-nearest neighbor queries over moving objects. In: ICDE. (2005) 631–642

8. Hu, H., Xu, J., Lee, D.L.: A generic framework for monitoring continuous spatial queries over moving objects. In: SIGMOD. (2005) 479–490
9. Mouratidis, K., Hadjieleftheriou, M., Papadias, D.: Conceptual partitioning: An efficient method for continuous nearest neighbor monitoring. In: SIGMOD. (2005) 634–645
10. Brinkhoff, T.: A framework for generating network-based moving objects. GeoInformatica **6**(2) (2002) 153–180