K-Nearest Neighbor Search for Moving Query Point

Zhexuan Song[1] and Nick Roussopoulos[2]

[1] Department of Computer Science
University of Maryland
College Park, MD 20742, USA
zsong@cs.umd.edu
[2] Department of Computer Science &
Institute For Advanced Computer Studies
University of Maryland
College Park, MD 20742, USA
nick@cs.umd.edu

Abstract. This paper addresses the problem of finding k nearest neighbors for *moving* query point (we call it k-NNMP). It is an important issue in both mobile computing research and real-life applications. The problem assumes that the query point is not static, as in k-nearest neighbor problem, but varies its position over time. In this paper, four different methods are proposed for solving the problem. Discussion about the parameters affecting the performance of the algorithms is also presented. A sequence of experiments with both synthetic and real point data sets are studied. In the experiments, our algorithms always outperform the existing ones by fetching 70% less disk pages. In some settings, the saving can be as much as one order of magnitude.

1 Introduction

One of the most important operations in a spatial database management system is the k-nearest neighbor search. The problem is: given a set S of n sites and a query point q, find a subset $S' \subseteq S$ of $k \leq n$ sites such that for any sites $p_1 \in S'$ and $p_2 \in S - S'$, $dist(q, p_1) \leq dist(q, p_2)$. This is also called *top-k selection query*. The k-nearest neighbor searching is used in a variety of applications, including knowledge discovery and data mining [6], CAD/CAM systems [10] and multimedia database [15].

In this paper, we study the following case: the query point q is a moving point. We call it "k-nearest neighbor search for moving query point" (K-NNMP). As the following example illustrates, the problem arises naturally in mobile computing environment.

Example [16] [Mobile E-commerce]: Provide the following service for a moving car: tell the driver where the nearest gas stations are. In the above example, the car is a moving query point which changes its location continuously.

C.S. Jensen et al. (Eds.): SSTD 2001, LNCS 2121, pp. 79–96, 2001.

Gas stations are sites in set S. The sites are static and their positions are known in advance. The service is to provide the query point with real-time reports about the k nearest neighbors.

A special case: 1-NNMP has been studied in Computational Geometry for quite a long time and the solution is trivial after Voronoi diagram is used. Given a site set S, the Voronoi diagram can be pre-computed in an efficient $O(n \log n)$ algorithm [5] where n is the number of sites in S. Then for a moving query point q, at each sampled position, the nearest neighbor can be quickly found in $O(\log n)$ time by locating the cell of Voronoi diagram that contains q. Unfortunately, the Voronoi solution cannot be extended to general k-NNMP problem because it is very difficult to generate a generalized k-Voronoi diagram.

Before we start to search for solutions, we need to find a way to represent the movement of the query point. In most cases, the trace of a moving query point is a continuous curve in working space and it can hardly be described by a function within reasonable level of complexity. Thus periodical sampling technique is widely used: the time period is divided by $n+1$ time stamps into n equi-length intervals. At each time stamp, which we called "sampled position", the location information of the query point is collected. The location of the query point between two consecutive sampled positions is estimated using linear or polynomial splines. In this paper, we adopt this approach because it provides a good approximation when the time intervals are short. Later in this paper, without confusion, we assume that sampled position t, time stamp t, and time t are exchangeable.

When periodical sampling technique is applied, in order to provide an acceptable solution for k-NNMP problem, we need to find correct k nearest neighbors at every sampled position. A naive solution is: to launch a new k-nearest neighbor search at each sampled position. Since the k-nearest neighbor search operation is relatively expensive, this solution is inefficient.

We propose a progressive approach in this paper. Our methods are motivated by the following observation: when two query positions q_1 and q_2 are close, the results of the k-nearest neighbor search S_1 and S_2 must be related. Once we have S_1, S_2 can be achieved with much less work. To authors' knowledge, the current research interest is mainly focus on designing a sophisticated index structure for sites so that the answer can be found quickly for any query points [8].

One assumption is made in this paper: the information of sites is provided before query and the sites are stored in a R-tree-family structure. R-tree and its variants [7] are excellent structures for indexing spatial data. The sites are treated as points in R-tree.

Our main contributions include:

- We review the k-NNMP problem from a new point of view. Instead of looking for an efficient spatial index structure for sites, at each sampled position, we try to utilize the information contained in the result sets at the previous

sampled positions. In some cases, we can extract the correct answer just from the history data and no further search is necessary.
- If the previous result set is obsolete, a new search directly from the sites is unavoidable. Our algorithms will provide a much better start point: the initial search bound is much lower than the pure static branch-and-bound algorithms. Thus, more nodes in the R-tree can be pruned without checking. In our experiments, we show that with this improvement, the total disk pages accessed by our algorithms are about 70% less than the existing ones.
- To obtain the first several result sites quickly is another interesting problem in the k-nearest neighbor search problem. In our progressive methods, the history result set is scanned first and those sites which, by our theorem, must be in the current result set are picked out. Therefore, in most cases, our algorithms find the first several result sites quickly.
- If the next position of the query point can be precisely predicted, we further optimize our algorithm by introducing another buffer. The second buffer stores the site candidates for the next query point position during the search procedure. Therefore, the initial search bound is even lower.

The organization of this paper is the following. In section 2, related work is introduced. In section 3 four algorithms are proposed. The experimental results are studied in section 4 and in section 5 conclusions and future research plans are presented.

2 Related Work

Recently, many methods have been proposed for k-nearest neighbor search in *static* environment. They can be classified into two categories depending on the total rounds of searches on the site set [14].

The first category is called single-step search. The methods in this category scan the site set only once.

Roussopoulos et. al. [13] proposed a branch-and-bound algorithm for querying spatial points storing in an R-tree. Meanwhile, they introduced two useful metrics: MINDIST and MINMAXDIST for ordering and pruning the search tree. The algorithm is briefly described as following.

- Maintain a sorted buffer of at most k current nearest neighbors.
- Initially set the search bound to be infinite.
- Traverse the R-tree, always lower the search bound to the distance of the furthest nearest neighbor in the buffer, and prune the nodes with MINDIST over the search bound, until all nodes are checked.

The performance of this algorithm was further investigated in [11] and the metrics were later used in Closest Pair Queries [4].

The other category is called multi-step search. The methods in this category scan the dataset multiple times until the proper radius is attained.

Korn, et. al. proposed an adapted version of multi-step algorithm [9]. The algorithm worked in several stages. First, a set of k primary candidates was selected based on stored statistics. After the candidate set was checked, an upper bound d_{max} was found, which guaranteed that at least k sites were within the distance d_{max} from the query point q. Next, a range query was executed on site set to retrieve the final candidates. The range query was: select all sites in the range R from the site set where $R = \{o|dist(o, q) \leq d_{max}\}$. The number of final candidates could not be less than k. At last, those final candidates were checked again and the final results were found.

Seidl and Kriegel further extended the method [14]. In their algorithm, every time a candidate site was checked, d_{max} was adjusted. They reported that their algorithm was optimal and the performance was significant improved.

Chaudhuri and Gravano [3] adopted the same approach. Their contribution was to use histograms in their algorithm to make the first d_{max} guess more accurate.

Finally, the possibility of using Voronoi cells in nearest neighbor search was studied [1] and BBD-tree for approximated search in fixed dimensions was proposed [8].

3 *K*-NNMP Problem

The following symbols are used in this section: q is the moving query point and q_t is the location of q at sampled position t. S is the site set which contains $n = |S|$ sites. At any time t, the sites in S are sorted in ascendant order based on their distance to q_t. The site which is ranked at position i has the distance $D_t(i)$. By definition, we have $D_t(1) \leq D_t(2) \leq \ldots \leq D_t(n)$. We also define ϵ_t to be a distance upper bound so that within this distance from q_t, we are guaranteed to find at least k sites. By definition, $D_t(k) \leq \epsilon_t$.

All our four algorithms and the naive solution which we mentioned in the first section are constructed on a static k-nearest neighbor search algorithm. In this paper, we pick the static branch-and-bound algorithm [13] because in this algorithm, no database statistics are required. So its query performance is foreseeable and stable.

Generally speaking, the moving pattern of objects (such as the pedestrians) is random rather than following some specific rules. Our first three algorithms are designed for the general cases. In some special cases, the next position of objects (such as an airplane) can be calculated. With the extra information, our methods can be further optimized. The fourth algorithm is for this purpose.

3.1 Fixed Upper Bound Algorithm

The naive solution launches a new search at every sampled position with static branch-and-bound algorithm and the search bound is initially set to be infinite. If we have no extra information, this step is essential for finding the correct answer. But in the k-NNMP problem, the search result at the previous sampled position may give us some clues for a smaller initial guess. It is an evident fact that with a smaller initial search bound, the static branch-and-bound algorithm becomes more efficient. In this part, we want to find a smaller search bound ϵ_{t+1} from the existing $D_t(i)$ where $1 \leq i \leq k$.

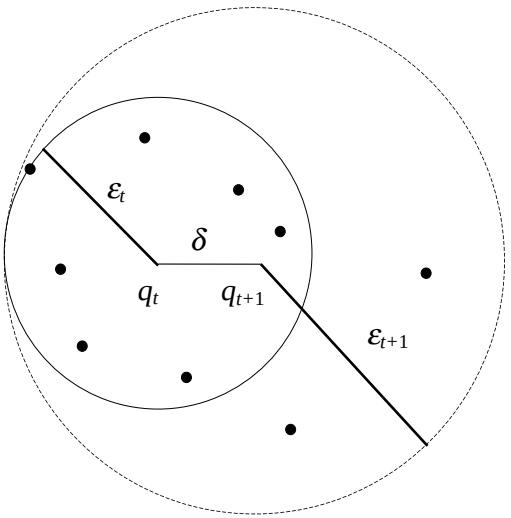

Fig. 1. ϵ_{t+1} does not need to be infinite

Look at Figure 1, at time t, the query point is at q_t. Within the solid circle, there are k sites. At time $t+1$, the query point moves to q_{t+1}, the distance between q_t and q_{t+1} is δ. All the sites whose distances to q_{t+1} are less than or equal to ϵ_{t+1} are in the dashed circle. By definition, this ϵ_{t+1} is legal if and only if within the dashed circle, there are at least k sites. This restriction can be easily met if the dashed circle encloses the solid circle. We have the following theorem.

Theorem 1. *Suppose at time t, the k nearest neighbors of query point at location q_t are $\{p_1, p_2, \ldots, p_k\}$ and $D_t(k)$ is the maximum distance of these sites to q_t. At time $t+1$, the query point moves to q_{t+1}. We claim that $\epsilon_{t+1} = D_t(k) + \delta$ is legal, where δ is the distance between q_t and q_{t+1}.*

Proof. By definition we have:

$$distance(p_i, q_t) \leq D_t(k) \qquad (\forall i \leq k)$$

According to triangular inequality, we have:

$$distance(p_i, q_{t+1}) \leq D_t(k) + \delta \qquad (\forall i \leq k)$$

which means that there are at least k sites whose distances to q_{t+1} are no more than $D_t(k) + \delta$. By definition, we know that $\epsilon_{t+1} = D_t(k) + \delta$ is legal.

The fixed upper bound search algorithm works as follow. At sampled position 1, after the routine static bound-and-branch search, the location of the query point is recorded as well as $D_1(k)$. Later at sampled position $t > 1$, after the position of query point q_t is obtained, the initial search bound is calculated according to the above theorem and a new static search is launched. After the search, the values of $D_t(k)$ and q_t are stored and prepared for the actions at the next sampled position.

3.2 Lazy Search Algorithm

In the fixed upper bound search algorithm, a new search is launched at each sampled position, even though the movement of the query point is too small during that time period to cause any changes in the result. In this part, we give a lazy search algorithm and show that in some cases a new search is unnecessary.

After observing most static k-nearest neighbor algorithms, we find the following fact: at time t, when we search for the k nearest neighbors, $D_t(k+1)$ can always be obtained for free or with little extra work.

For example, in the adapted multi-step k-nearest neighbor search algorithm [9], at the second round, most of time, the result of range query contains more than k elements which need further check at the last step. To get the value of $D_t(k+1)$, we simply record the distance of the first follow-up loser. Look at another example, in static bound-and-branch algorithm [13], if we maintain a sorted buffer with size $k+1$ instead of k, after the whole R-tree is scanned, the value of $D_t(k+1)$ is the distance between the query point and the site which we stored in the extra buffer.

With the knowledge of $D_t(k+1)$, we have the following theorem:

Theorem 2. *Suppose at time t, the k nearest neighbors of query point at location q_t are $\{p_1, p_2, \ldots, p_k\}$, $D_t(k)$ is the maximum distance of these sites to q_t and $D_t(k+1)$ is the minimum distance of the sites outside that set. At time $t+1$, the query point moves to q_{t+1}, we claim that the result set will be the same if*

$$\delta \leq \frac{D_t(k+1) - D_t(k)}{2}$$

where δ is the distance between q_t and q_{t+1}.

Proof. By definition we have:

$$distance(p_i, q_t) \leq D_t(k) \qquad (\forall i \leq k)$$

After the query point moves to q_{t+1}, by triangular inequality, we have:

$$distance(p_i, q_{t+1}) \leq D_t(k) + \delta \qquad (\forall i \leq k)$$

For any site p not in this set, by triangular inequality, we have:

$$D_t(k+1) - \delta \leq distance(p, q_{t+1})$$

When $\delta \leq \frac{D_t(k+1)-D_t(k)}{2}$, for any site p not in this set, we have:

$$distance(p_i, q_{t+1}) \leq D_t(k) + \delta \leq D_t(k+1) - \delta \leq distance(p, q_{t+1}) \qquad (\forall i \leq k)$$

That means $\{p_1, p_2, \ldots, p_k\}$ is still the correct result set.

A very intuitive fact is illustrated in the above theorem: when the movement of the query point is small, the query result will not change.

Another important issue in static nearest neighbor query is to get the first solution quickly. In our progressive algorithm, even though we have to start a new search, some sites in the previous result set will still appear in our new result. It makes us to think about the following problem: which part in the previous result will still be in the new search result after the query point moves a distance δ? The following theorem is a solution.

Theorem 3. *Suppose at time t, k nearest neighbors to query point at location q_t are $\{p_1, p_2, \ldots, p_k\}$, $D_t(k)$ is the maximum distance of these sites to q_t and $D_t(k+1)$ is the minimum distance of the sites outside that set. At time $t+1$, the query point moves to q_{t+1}, we claim that the new result still contains sites p_i with $D_t(i) < D_t(k+1) - 2\delta$, where δ is the distance between q_t and q_{t+1}.*

Proof. From the previous theorems, we know that at time $t+1$, $D_{t+1}(k+1) \geq D_t(k+1) - \delta$. For each site p_i in the result set, by triangular inequality, we have $dist(q_{t+1}, p_i) \leq D_t(i) + \delta$. Put these two together, if $D_t(i) < D_t(k+1) - 2\delta$ holds, we have

$$distance(q_{t+1}, p_i) \leq D_t(i) + \delta < D_t(k+1) - \delta \leq D_{t+1}(k+1)$$

which concludes the proof.

Here is the lazy search algorithm. At sampled position 1, the following information is maintained: the correct query result set, the position of query point, $D_1(k)$ and $D_1(k+1)$. Later at sampled position $t > 1$, the first step is to check whether the buffered query result is still correct (theorem 2). If so, return the result. Otherwise, find the "still-correct" sites (theorem 3). Then calculate the initial ϵ_t (theorem 1) and start a static search. After the new result set is found, store the result set into our buffer, along with the value of $D_t(k)$, $D_t(k+1)$ and q_t; and wait for the next query.

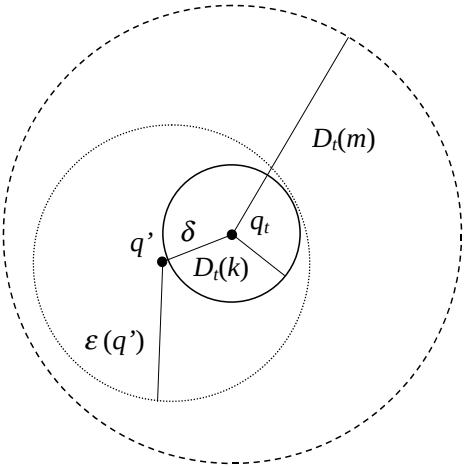

Fig. 2. In some cases, there is no need to update the buffer

3.3 Pre-fetching Search Algorithm

Pre-fetching technique is widely used in disk-bounded environments. Comparing with the lazy search algorithm, this time, we go one step further. We execute m-nearest neighbor search at some sampled positions where $m > k$, the results are stored in a buffer which is maintained in memory. At the remaining sampled positions, we only check the buffer information and do not search the disk.

One problem remains for this algorithm: since the buffer stores the m nearest neighbors at previous query position, when we should updated the contents in the buffer? We propose the following theorem as a solution.

Theorem 4. *Suppose at time t, the m nearest neighbors of query point at location q_t are stored in the buffer, where $m > k$. $D_t(k)$ and $D_t(m)$ are the k-th and m-th distance among those sites. Later the query point moves to a new location q'. We claim that there is no need to update the contents in the buffer if*

$$\delta \leq \frac{D_t(m) - D_t(k)}{2}$$

where δ is the distance between q_t and q'.

Proof. As indicated by theorem 1, when query point is at q', $\epsilon(q')$ is valid if $\epsilon(q') \geq D_t(k) + \delta$. Look at Figure 2, the information of the sites which are in the dashed circle is stored in our buffer. It is clear that there is no need to update the contents in the buffer if and only if the dashed circle contains the doted circle. That is: $\epsilon(q') \leq D_t(m) - \delta$. Combine this with the conclusion in theorem 1, we have

$$D_t(k) + \delta \leq \epsilon(q') \leq D_t(m) - \delta$$

i.e.

$$\delta \leq \frac{D_t(m) - D_t(k)}{2}$$

Theorem 4 tells us when the contents in the buffer are obsolete and required to be updated. Observing the conclusion, we find that the lazy search algorithm is a special case of this algorithm if we set $m = k + 1$. There is no corresponding theorem about first solution in this algorithm. But it would be very helpful if we inspect the buffer and check the sites whose distances to q' are less than or equal to $\frac{D_t(m)-\delta}{2}$ before a new search.

Here are the details of the pre-fetching search algorithm: at sampled position 1, an extended m nearest neighbor search algorithm is executed. The results are stored in a buffer. Among them, the k nearest neighbors are identified. The information about $D_1(k)$, $D_1(m)$ and the query position are stored too. Later at sampled position $t > 1$, after the position of query point q_t is obtained, our first check is whether the results are in our buffer (theorem 4). If the answer is yes, we only need to examine the sites in the buffer; if the answer is no, a new static m-nearest neighbor search is launched. After the search, the value of $D_t(k)$, $D_t(m)$ and q_t are stored and prepared for actions at later sampled positions.

3.4 Dual Buffer Search

In some applications, the next position of the query point can be partly predicted. For example, in an air-traffic control system, the moving query point is an airplane. At any given time t, q_{t+1} can be calculated based on the speed of object and the length of time interval. However, the prediction of q_{t+1} cannot be 100% correct because of many unforeseeable reasons.

In this paper, we adopt the assumption about the worst-case sampling error of the prediction of the position made in [12]: suppose at sampled position t, the query point is at q_t. The prediction of the position is a pair of data (q'_{t+1}, r) so that the next position q_{t+1} has the same possibility to be within the circle around q'_{t+1}, and r is the radius. Here q'_{t+1} is called "predicted next position" of the query point and r is called "uncertainty factor".

Here is the basic idea of this algorithm. If during the search procedure at time t, for each site candidate, we also check the distance from the site to q'_{t+1} and store the k best sites in another buffer, then at the next sampled position $t + 1$, we may have a better start.

The details of the dual buffer search algorithm is as following. There are two buffers with size k in the system. The first buffer is for the site candidates of the current position and the second one is for the candidates of the next position. At sampled position 1, we do a routine k-nearest neighbor search. Meanwhile, for each site we check, we also calculate the distance of the site to the predicted next position of query point q'_2. Among all the checked sites, the k current nearest neighbors are maintained in the first buffer and the k best results to q'_2 are stored

in the second buffer. Later at sampled position $t > 1$, the sites in the second buffer are moved to the first buffer and the maximum distance of the sites to q_t in the buffer is used as the initial search bound. At the same time, in the second buffer, the k sites are still kept. This time, they are sorted by the distances to the new predicted next query position q'_{t+1}. Then, a new k-nearest neighbor search is launched and both buffers are updated.

With the second buffer, the initial search bound may be lowered.

Theorem 5. *In the dual buffer search algorithm, at any sample position t, the real initial search bound (the maximum distance of the sites in the second buffer to q_t) is no greater than r plus the predicted initial search bound (the maximum distance of the sites in the second buffer to q'_t). If $r = 0$, at all sampled positions, the initial search bound in the dual buffer search algorithm is always no greater than that in the fixed upper bound algorithm.*

The proof is obvious and is omitted in this paper. In real life, r cannot always be zero. If r is large, the real initial search bound may be even worse than that in the fixed upper bound algorithm. In the experiment section, we will use an experiment to show the impact of r.

3.5 Discussion

In this part, we give some discussion about the parameters that affect the performance of our four algorithms: the fixed upper bound algorithm, the lazy search algorithm, the pre-fetching search algorithm and the dual buffer search algorithm.

The fixed upper bound algorithm always launches a search at new sampled position with a better initial search bound. This guarantees that in any situation, it works better than the static solution because more nodes are pruned in the algorithm. The dual buffer search algorithm adopts the similar idea. If the prediction of the next position is precise, the initial search bound will be even lowered.

The cost of a new search in the lazy search algorithm is a little more expensive than that in the fixed upper bound algorithm. The reason is mainly because in the lazy search algorithm, extra information, $D(k + 1)$, is collected. For the same reason, a new search in the pre-fetching search is even more expensive. Fortunately, we do not have to launch a new search at every sampled position in both algorithms. According to our theorems, at some sampled positions, using the information in our buffer is enough. We call these positions "no search positions". It is obvious that the more "no search positions" we have, the better performance the algorithms reach.

At least three factors affect the performance of the lazy search algorithm: number of sites, speed of the moving query point and k. Here we assume that the sites are distributed uniformly.

According to theorem 2, a sampled position is a "no search position" if during the time period, query point moves a small distance $\delta \leq \frac{D_t(k+1)-D_t(k)}{2}$. When the number of sites is large, $D_t(k+1) - D_t(k)$ will be small; when the speed of the moving query point is fast, δ will be big. In both cases, the number of "no search positions" decreases.

The relation between k and the performance is a little more complicated. By definition, we know that $D_t(i)$ is a non-decrease sequence. At the same time, we must also point out that when sites are in uniform distribution, $D_t(k)$ is proportional to $\frac{1}{\sqrt{k}}$. It means that the value of $D_t(k+1) - D_t(k)$ reduces as k grows. So does the number of "no search positions".

Besides the above three parameters, another one that is important for the pre-fetching search algorithm is the buffer size. There is a tradeoff here: maintaining a small pre-fetched buffer lowers the cost for individual disk search, and maintaining a large one generates more "no search positions". In the next section, we will use some experiments to demonstrate the impact of the buffer size.

In sum, the progressive methods are better than static methods. Among them, the lazy search algorithm and the pre-fetching search algorithm have better performance for small site number, slow moving speed of query point, and little k. This will be showed in our experiments.

4 Experimental Result

To access the merit, we performed some experimental evaluations of different methods on both synthetic and real data sets.

4.1 Experiment Settings and Methods Selection

In our experiments, we use a unit square $[0, 1)^2$ as working space. In both synthetic and real data sets, the sites are normalized into the unit space. All sites are organized in a R-tree [7].

Along with the naive solution and our four algorithms, we also test the query performance of the brutal search algorithm. We choose brutal search algorithm because it is the only method available if the sites are not indexed. In the pre-fetching search algorithm, we select three different buffer sizes: twice, five times and ten times as much as the result size.

The abbreviations of algorithms used in this part are showed in Table 1.

Experiments were performed on a Sun SPARCstation 5 workstation running on Solaris 2.7. The workstation has 128 M Bytes of main memory and 4 G Bytes of disk. The page size is 4 K Bytes for both disk I/O and R-tree nodes. In real-world data sets, the information of each site consists of its id, name, location, etc. The size of each record is 100 bytes, which leads to 40 points per disk page. In synthetic data sets, the size of each data is 20 bytes (4 bytes for its id and 8 bytes for each dimension) which lead to 200 points per disk page. The experiments were written in JAVA 2.0

Table 1. Algorithm abbreviations

BS	Brutal Search Algorithm
SBB	Static Branch-and-Bound Algorithm (naive solution)
FUB	Fixed Upper Bound Algorithm
LS	Lazy Search Algorithm
PS-m	Pre-fetching Search Algorithm with buffer size $m \times k$ (For example, PS-2 means the buffer size we choose is twice as much as k.)
DBS	Dual Buffer Search algorithm

4.2 Data Sets

The real-world data sets we used come from TIGER [2]. The particular three data sets were 120 hospitals, 1982 churches and 1603 schools in Maryland, USA. Figure 3, 4 and 5 show the site distribution of each set.

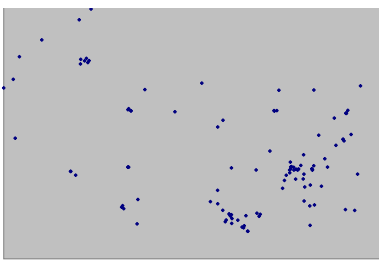

Fig. 3. Hospital distribution

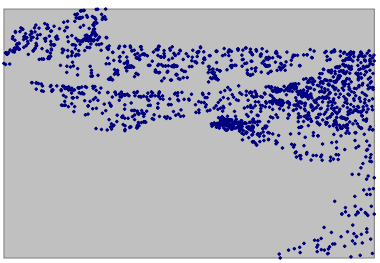

Fig. 4. Church distribution

The synthetic data set contains a large number of uniformly distributed sites in working space.

In our experiments, we measure the performance of various algorithms based on the number of disk pages they accessed. In most cases, nearest neighbor search problem is disk-bounded and the computational cost is trivial comparing with the I/O cost. The average number of disk pages each method accesses at every sampled position provides a direct indication of its I/O performance. For LS, PS-2, PS-5 and PS-10, we also record the percentage of "no search position". This one provides useful information about the reason why LS and PS-m work well in some cases.

4.3 Query Sets

The moving queries are generated as following. First randomly pick a point in working space as the start point. Then we define a speed for the moving object.

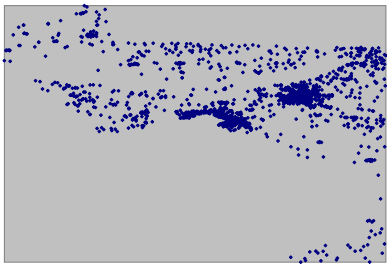

Fig. 5. School distribution

The moving object keeps that speed but with a 10% error. For example, if the defined speed is 0.001, the actual speed of the moving object is randomly picked from 0.0009 to 0.0011. At each sampled position, we calculate the current position of the moving object and execute the algorithms. In each continuous query, the time is divided into 1,000 intervals. The results we listed are the average values of more than 100 queries.

4.4 Experimental Results Using Synthetic Data Sets

We use the synthetic data sets to study the parameters that affect the performance of algorithms: r, site number, speed of moving object and k.

In the first experiment, we want to compare the initial search bound in FUBS and DBS with different "uncertainty factor" r. To make the result clear, we only display the ratio numbers (i.e. the initial search bound / $D(k)$). The lower the ratio number is, the better the algorithm will be. The other parameters are fixed: $k = 10$, speed $= 0.0001$ and the site number is 100,000. r varies from 0 to $0.5 \times \delta$ where δ is the actual distance between two consecutive query positions. In figure 6, y-axis is the ratio number and x-axis is the value of r/δ.

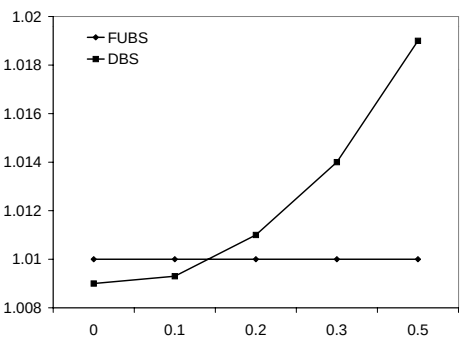

Fig. 6. The initial search bound of FUBS and DBS with different r

Figure 6 shows that DBS has smaller initial search bound only when the value of r is low (less than 15% of δ). As r grows, i.e. the prediction becomes not very precise, the performance of DBS is worse than FUBS. That means if the movement pattern of the object is random-like, the prediction information is not helpful but has negative impact on the performance. In the following experiments concerning DBS, we fix r to be $0.1 \times \delta$.

In the second experiment, we vary the site number from 1,000 to 1,000,000 to compare the performance of different algorithms. The other two parameters are fixed: $k = 10$ and speed $= 0.0001$.

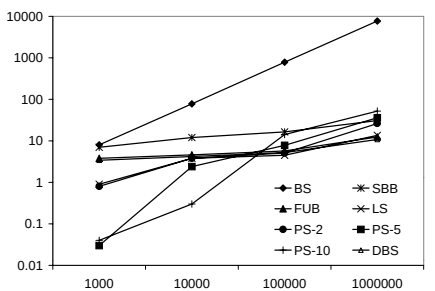

Fig. 7. Disk page access vs. site number

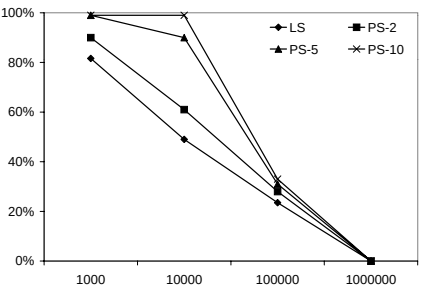

Fig. 8. Percentage of no search position vs. site number

In Figure 7, x-axis is the number of sites we indexed for the queries and y-axis is the average pages accessed for each method at every sampled position. In Figure 8, y-axis is the percentage of "no search positions" for four algorithms with pre-fetching information.

The following facts are showed in Figure 7 and Figure 8.

1. R-tree grows when the site number increases. All methods require checking more disk pages for larger site number.
2. FUB always outperforms SBB because of its good initial search bound.
3. Due to the lower initial search bound, DBS always beats FUBS by checking about 10% less nodes.
4. The percentage numbers of "no search position" for LS, PS-2, PS-5, and PS-10 are more than 80% when the site number is 1,000. Among them, the numbers of PS-5 and PS-10 are close to 100%. The outcome is that their performances are much better than other three opponents.
5. As the site number increases, the percentage numbers drop quickly. When the site number is 1,000,000. The numbers of all four algorithms become 0%. That means the information we pre-fetched is totally useless. At this moment, the bigger buffer size we choose, the worst performance it becomes. PS-10 and PS-5 even access more disk pages than the naive solution.

6. For the pre-fetching search algorithms with large buffer size (PS-5 and PS-10), the performance easily becomes very bad if the percentage of "no search position" decreases. It means that choosing large buffer size for pre-fetching search algorithm sometimes may be very dangerous.

In the next experiment, we vary k from 1 to 100 and fix site number to be 100,000 and speed to be 0.0001. The result is showed in Figure 9 and 10. x-axis is the value of k.

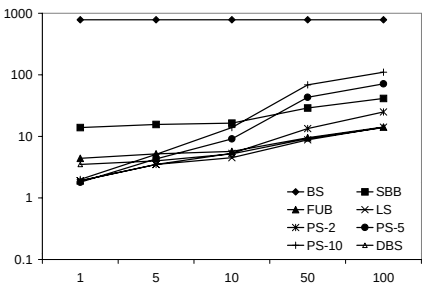

Fig. 9. Disk page access vs. k

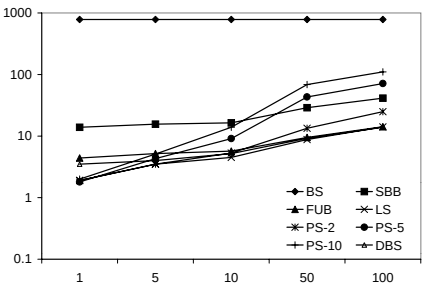

Fig. 10. Percentage of no search period vs. k

Generally, all methods except BS check more disk pages when k grows, and FUB always outperform SBB by a factor of 3. DBS has similar curve as FUB but the corresponding value is about 10% less. LS and three PS-m algorithms follow the same pattern as in the first experiment. The little difference is the percentage number is not exactly 0 when $k = 100$. In three pre-fetching search algorithms, increasing buffer size brings very little benefit when $k = 100$. (Double the buffer size from PS-5 to PS-10 only adds 0.2% to the percentage of "no search position".) It is worth noting that even with such a small percentage number, LS still outperforms FUB.

Finally, we vary the query point speed from 0.00005 to 0.001 and fix the point number to be 100,000 and k to be 10. x-axis is the value of speed.

As Figure 11 and 12 show, the performance pattern remains to be the same. This time, the percentage of "no search position" drops much faster. PS-10 and PS-5 win only when the speed of query point is very slow, and lose to most algorithms in the rest cases.

In conclusion, FUB, DBS and LS always outperform the pure static method by a factor of 3 except for very small site number. When the prediction of the next position of the query point is precise, the extra information helps DBS to beat FUB by accessing about 90% of disk pages. LS is better than FUB in most cases when site number is not too large and speed of query point is not too fast. PS-m has the best performance when the percentage of "no search position" is large, but lost to FUB and LS in most situations. The pre-fetching algorithms

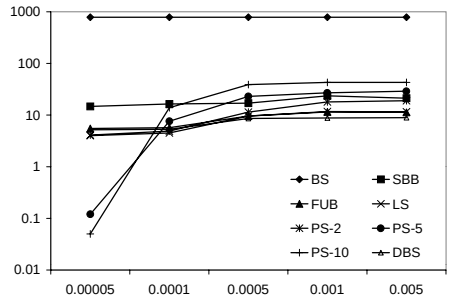

Fig. 11. Disk page access vs. speed

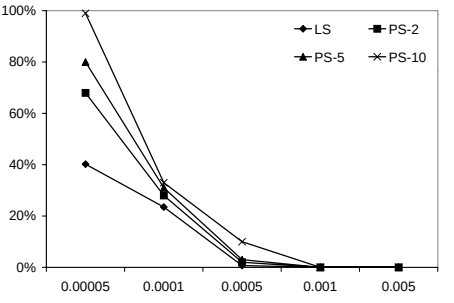

Fig. 12. Percentage of no search period vs. speed

with large buffer size are more sensitive to parameter change than ones with small buffer size.

4.5 Experimental Results Using Real Data Sets

In real-world data sets, we want to simulation the following situation: a car moves at a normal speed (65 mph or 105 km per hour with 20% error) in Maryland, USA. The route of the car is randomly selected. A real-time report about the 10 nearest sites is provided and the time interval is one minute.

In the experiments, we test three data sets separately and in the fourth experiment, we put these site data together.

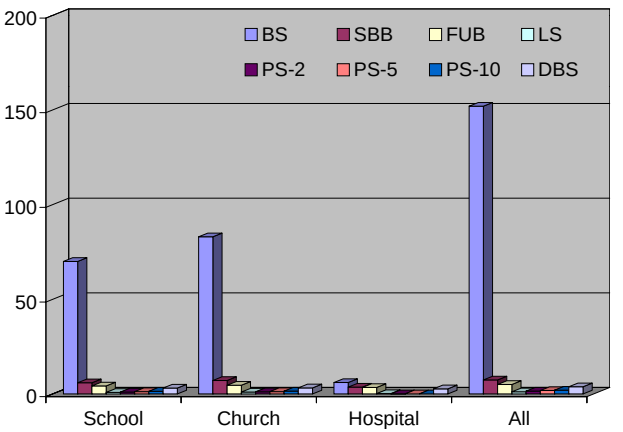

Fig. 13. Disk page access of different methods in four data sets

Figure 13 compares the number of R-tree nodes fetched from disk by each algorithm at every sampled position. In the hospital dataset, since the site num-

ber is too small (only about 100), the improvement of our algorithms is not very clear. In the rest three cases, the proposed FUB, LS, PS-m and DBS algorithms access a much smaller number of R-tree nodes than SBB and BS methods.

5 Conclusion

We studied the k-nearest neighbor search problem on moving query point and proposed progressive methods which minimize the query operation by using information of previous query and pre-fetched results. Our idea is easy to be implemented and can be combined with any single-pass or multiple-pass static nearest neighbor search algorithms, although we use static branch-and-bound algorithm in this paper. All the techniques we presented boost the query performance greatly.

In this paper, the blind pre-fetching algorithm buffers a lot of useless points. If we have the information about the path of the query point, how to efficiently buffer the pre-fetch results is a challenging problem.

References

1. S. Berchtold, B. Ertl, D. Keim, H. Kriegel, and T. Seidl. Fast nearest neighbor search in high-dimensional space. In *Proceedings of International Conference on Data Engineering*, Orlando, USA, 1998.
2. U. C. Bereau. Tiger(r) topologically integrated geographic encoding and referencing system, 1998.
3. S. Chaudhuri and L. Gravona. Evaluating top-k selection queries. In *Proceedings of International Conference on Very Large Database*, Edinburgh, Scotland, 1999.
4. A. Corral, Y. Manolopoulos, Y. Theodoridis, and M. Vassilakopoulos. Closest pair queries in spatial databases. In *Proceedings of ACM SIGMOD International Conference on Management of Data*, Dallas, USA, 2000.
5. M. de Berg, M. van Dreveld, M. Overmars, and O. Schwarzkop. *Computational Geometry: Algorithms and Applications*. Springer-Verlag, 1997.
6. U. Fayyad, G. Piatetsky-Shapiro, P. Smyth, and R. Uthurusamy. *Advances in Knowledge Discovery and Data Mining*. AAAI Press/MIT Press, 1996.
7. V. Gaede and O. Guenther. Multidimensional access methods. *ACM Computer Surveys*, 30, 1998.
8. T. Kanungu, D. Mount, N. Netanyahu, C. Piatko, R. Silverman, and A. Wu. Computing nearest neighbors for moving points and applications to clustering. In *Proceedings of 10th ACM-SIAM Symposium on Discrete Algorithms*, 1999.
9. F. Korn, N. Sidiropoulos, C. Faloutsos, E. Siegel, and Z. Protopapas. Fast nearest neighbor search in medical image databases. In *Proceedings of International Conference on Very Large Database*, Mumbai, India, 1996.
10. H. Kriegel. S3: Similarity search in cad database systems. In *Proceedings of ACM SIGMOD International Conference on Management of Data*, Tucson, USA, 1997.
11. A. Papadopoulos and Y. Manolopoulos. Performance of nearest neighbor queries in r-trees. In *Proceedings of International Conference on Database Theory*, Delphi, Greece, 1997.

12. D. Pfoser and C. Jensen. Capturing the uncertainty of moving-object representations. In *Proceedings of International Symposium on Large Spatial Databases*, 1999.
13. N. Roussopoulos, S. Kelley, and F. Vincent. Nearest neighbor queries. In *Proceedings of ACM SIGMOD International Conference on Management of Data*, San Jose, USA, 1995.
14. T. Seidl and H. Kriegel. Optimal multi-step k-nearest neighbor search. In *Proceedings of ACM SIGMOD International Conference on Management of Data*, Seattle, USA, 1998.
15. T. Seidl and H. Kriegel. Efficient user-adaptable similarity search in large multimedia database. In *Proceedings of International Conference on Very Large Database*, Athens, Greece, Athens.
16. P. Sistla and O. Wolfson. Research issues in moving objects database. In *Proceedings of ACM SIGMOD International Conference on Management of Data*, Dallas, USA, 2000.