Processing Constrained *k*-Closest Pairs Queries in Crime Databases

Shaojie Qiao, Changjie Tang, Huidong Jin, Shucheng Dai, Xingshu Chen, Michael Chau, and Jian Hu

Abstract Recently, spatial analysis in crime databases has attracted increased attention. In order to cope with the problem of discovering the closest pairs of objects within a constrained spatial region, as required in crime investigation applications, we propose a query processing algorithm called Growing Window based Constrained *k*-Closest Pairs (GWCCP). The algorithm incrementally extends the query window without searching the whole workspace for multiple types of spatial objects. We use an optimized R-tree to store the index entities and employ a density-based range estimation approach to approximate the query range. We introduce a distance threshold with regard to the closest pair of objects to prune tree nodes in order to improve query performance. Experiments discuss the effect of three important factors, i.e., the portion of overlapping between the workspaces of two data sets, the value of *k*, and the buffer size. The results show that GWCCP outperforms the heap-based approach as a baseline in a number of aspects. In addition, GWCCP performs better within the same data set in terms of time and space efficiency.

Shaojie Qiao
School of Information Science and Technology, Southwest Jiaotong University, Chengdu 610031; Southwest Jiaotong University
e-mail: `qiaoshaojie@gmail.com`

Changjie Tang, Shucheng Dai, Xingshu Chen, Jian Hu*
School of Computer Science, Sichuan University, Chengdu 610065, China
e-mail: `{cjtang,daishucheng,chenxsh}@scu.edu.cn;`
*`hujianlucky@163.com`

Huidong Jin
Mathematical and Information Sciences, CSIRO, ACT 2601, Australia
e-mail: `warren.jin@csiro.au`

Michael Chau
School of Business, The University of Hong Kong, Pokfulam, Hong Kong, SAR
e-mail: `mchau@business.hku.hk`

An earlier version of the paper received the best paper award at the IEEE International Conference on Intelligence and Security Informatics 2008.

Keywords Spatial analysis · Crime databases · Constrained closest pairs · Query processing · R-tree

1 Introduction

Crime databases have developed into an important tool in crime investigate applications. In general, a "crime database" is defined as a spatial database that stores incident-based data focusing on the unique characteristics of a criminal incident that captures detailed crime characteristics, e.g., location, modus operandi, and time [1]. The locations of crime incidents illustrated in a crime-based map are represented by 2D points or 3D geometric entities.

Query processing for crime data has recently become popular, since the tragic events of September 11 and the subsequent anthrax contamination of letters produced a great effect on many aspects of society [2]. Crime investigators and anti-terrorism specialists typically maintain a spatial database storing the locations, distances, time, and other relevant information of crimes. For example, given the description of a fat suspect with long beard, a round face, and brown hair, an investigator is apt to find the closest pairs of spatial objects such as subways and airports where this criminal frequently visits. It is of practical value for law enforcement agencies to develop an efficient spatial query processing method in crime databases for predicting the potential crimes and preventing future crimes.

In addition, crime authorities can employ crime databases to allocate new facilities more appropriately. For instance, emergency calls to an emergency call center are dialed from various locations. For each call, a staff can generate a spatial event associated with an accident and dispatch ambulances to the scene of accident that is nearest to them. Another example is that the police station of New York City used its GIS to locate facilities and to respond to emergencies such as the attack on the World Trade Centers, in which location-based service plays an essential role in response and recovery efforts [3].

As for security domain, it is important to discover the closest pairs of multiple types of objects (i.e., objects derived from distinct spatial data sets) in crime databases. For example, a policeman may need to find one or multiple closest pairs of roads and crime scenarios, instead of the closest pairs of roads, in an efficient and effective manner.

A very common spatial query is the "*k*-Nearest Neighbor Query" (*k*NN) [4]. For example, a staff may want to find the *k* ($k \geq 1$, is the number of nearest neighbors) police officers closest to a crime scenario where a crime with *k* injured people occurred. The problem of "*k*-Closest Pairs Query" (*k*-CPQ) is an extension by combining nearest neighbor query with spatial join in order to find the *k*-closest pairs of spatial objects from two distinct data sets [5]. For example, anti-terrorist officers may be interested in finding the closest five supermarkets and banks, or the closest four bus stops and railways.

In this study, the *k*-CPQ with a spatial constraint is called constrained *k*-closest pairs query (*k*-CCPQ) [6]. This problem is of great practical value when applied to the security informatics domain of crime databases due to the following reasons.

- Typically, police officers often care for the query results within a given area which can help save query time instead of the whole space in the real-life scenarios.
- *k*-CCPQs may ask for *k*-closest pairs of any two types of objects. It is more appropriate for crime databases, since crime databases consist of multiple types of objects. For instance, the *k*-CCPQs in crime databases discovering the *k* pairs of subways and police departments resort to find the *k*shortest distances so that the police officer can efficiently dispatch police officers to the subways which are the possible targets of terrorist attack. In other words, it can assist in making an optimal allocation of police resources.

In order to solve the *k*-CCPQ problem in crime databases, we make the following contributions in this study:

1. We propose a novel constrained *k*-closest pairs query processing algorithm based on growing windows in crime databases, namely GWCCP. The window extends incrementally and terminates when discovering the *k*-CCPQs, which help eliminate the unnecessary distance calculations between spatial objects of crime incidents or other point locations.
2. We employ an optimized R-tree index structure to store the index entities and treat the distance of the maintained closest pair as a threshold. The closest pair whose distance is greater than this threshold is pruned which benefits reducing the response time of tracing criminals.
3. We use a density-based range estimation method to compute the square query range. It has some advantages. First, the space required to store the density information takes only several bytes. Second, every time a new range estimate is required, it is derived from the density of the previous window for query.
4. We conduct experiments to compare the proposed approach with the heap-based algorithm for *k*-CCPQ over two distinct crime data sets, and the results show that our algorithm performs well in most cases. In particular, the performance of GWCCP is better than SRCP-tree [7] within the same data set.

The rest of this chapter is organized as follows. Section 2 surveys the related work. Section 3 describes the problem of *k*-CCPQ and presents some useful metrics. Section 4 introduces the density-based range estimation method and proposes the algorithm of *k*-CCPQ query. Section 5 presents the performance studies of the proposed algorithm and discusses the experimental results. Finally, Section 6 concludes this chapter with a summary and directions for future work.

2 Related Work

The closest pairs query problem has unleashed a new wave of applications in the research of spatial databases [4–6]. However, there is relatively little work that is relevant to the closest pairs query processing problem in crime databases that is important to the security informatics domain.

Much work on the closest pair query problem focuses on applying R-tree to *k*NN queries, because R-tree is an efficient spatial index structure of retrieving data items based on the location of each object [8].

Roussopoulos [9] proposed an R-tree algorithm for *k*NN queries. The disadvantage of the algorithm is that once a node is visited, all nodes in its sub-tree have to be visited as well.

To avoid direct accesses to spatial indices, Liu et al. [10] transformed a *k*NN query into more window queries.

Hjaltason and Samet [11] proposed two spatial join operations between two R-tree indices. However, the approach is still required to store every pair of index entries and spatial objects in a priority queue. Thus, it still cannot avoid a large number of disk accesses.

Corral proposed an improved approach known as the Heap algorithm [4]. But, this approach is not efficient when there is much overlap between two spatial data sets, and this study does not take into account the case of *k*-CPQ with spatial constraints.

Ferhatosmanoglu et al. [12] applied some methods to answer the constrained nearest neighbor queries. However, the proposed algorithms are not suitable for *k*-CCPQ query.

The most similar work to this study has been explored by Shan. He made a good attempt to solve the *k*-CCPQ problem and proposed two kinds of SRCP-tree (Self Rang Closest Pair tree) [7]. However, SRCP-tree cannot support the CP (Closest Pair) query for multiple types of spatial objects.

To cope with the *k*-CCPQ problem in crime databases, we will introduce an efficient *k*-CCPQ query processing algorithm in the following section. This method can be applied to other spatial databases as well.

3 Problem Description

In this study, the *k*-CCPQ problem in crime databases is to seek *k* pairs of crime sites in two distinct data sets, and the sites located within a given spatial constraint. The formal definition is shown below [4].

Definition 1 (*k*-CCPQ) Given two spatial data sets, $S = \{s_1, s_2, \ldots, s_M\}$ and $T = \{t_1, t_2, \ldots, t_N\}$, be stored in two R-trees T_M and T_N, respectively. The *k*-CCPQ of S and T with regard to a given spatial constraint R is defined as *k*-ordered pairs as follows:

$$(s_{l_1},t_{h_1}),(s_{l_2},t_{h_2}),\ldots,(s_{l_k},t_{h_k})$$

where $s_{l_1},s_{l_2},\ldots,s_{l_k} \in S,t_{h_1},t_{h_2},\ldots,t_{h_k} \in T$, and each $s_i \in S$ and $t_i \in T$ has similar characteristics, respectively, $(s_{l_1},s_{l_2},\ldots,s_{l_k})$ and $(t_{h_1},t_{h_2},\ldots,t_{h_k})$ are inside R, such that:

$$\text{dist}\left(s_i,t_j\right) \geq \text{dist}\left(s_{l_k},t_{h_k}\right) \geq \text{dist}\left(s_{l_{k-1}},t_{h_{k-1}}\right) \geq \ldots \geq \text{dist}\left(s_{l_1},t_{h_1}\right)$$

$$\forall\left(s_i,t_j\right) \in (S \times T\text{-}\{\left(s_{l_1},t_{h_1}\right),\left(s_{l_2},t_{h_2}\right),\ldots,\left(s_{l_k},t_{h_k}\right)\}).$$

The *k*-CCPQs from the Cartesian product of *S* and *T* within *R*are *k* pairs that have the shortest distances between all pairs of points that are formed by selecting one point from *S* and the other point from *T*. Although "dist" stands for Euclidean distance in this study, the proposed method can be easily adapted to Minkowski distance as well. Here, we will give the useful metrics to measure the distances between two spatial objects.

Given two MBRs for *S* and *T*, two spatial objects $s_i \in S$ and $t_j \in T$. Following [4], MinMinDist(*S*, *T*) is the shortest distance between *S* and *T* boundaries and defined as:

$$\text{MinMinDist}(S,T) = \min\{\text{MiDist}(s_i,t_i)\} \tag{1}$$

where MinDist(s_i, t_i) represents the shortest distance between s_i and t_i.

MaxMaxDist(*S*, *T*) is the maximum distance between two points falling on *S* and *T* boundaries. MinMaxDist(*S*, *T*) is the minimum distance that guarantees that there is at least one pair of objects with distance smaller or equal to MinMaxDist [4]. They are defined below [4].

$$\text{MaxMaxDist}(S,T) = \max\{\text{MaxDist}(s_i,t_i)\} \tag{2}$$

$$\text{MinMaxDist}(S,T) = \min\{\text{MaxDist}(s_i,t_i)\} \tag{3}$$

where MaxDist(s_i, t_i) represents the maximum distance between s_i and t_i.

In crime databases, the MBRs of two data sets frequently overlap, because the criminals often commit an offense in a similar location. It is evident that the higher the portion of overlapping between two MBRs, the higher the probability that more pairs with small distances appear [4]. So, we propose a *k*-CCPQ processing algorithm especially suitable for handling the case of higher overlap between two data sets in this study.

4 *k*-CCPQ Processing Based on Growing Windows

This section introduces a new algorithm for *k*-CCPQ in crime databases called GWCCP by combining a new R-tree derived from SRCP-tree [7] to store the index entities and a density-based range estimation method [10]. The tree structure used in this paper is called C-tree. The main differences from the existing R-tree based index structures, such as Guttman's linear and quadratic R-tree [8], are on the following aspects [6]: (a) each index entry i is augmented with a triple (r_1, r_2, *dist*), where r_1 and r_2 are the closest pair of objects in the sub-tree rooted by i and *dist*is the distance between r_1 and r_2; (b) C-tree uses the Least Recently Used (LRU) buffer policy [13] in spatial selection, and spatial join between distinct data sets.

The spatial constraint is represented by $win = [(x_l, y_l); (x_u, y_u)]$, where (x_l, y_l) and (x_u, y_u) are the lower-left and upper-right corners of the spatial constraint. In this study, each object is represented by a 2D point.

Definition 2 (*k*-CCPQ on Windows) Given a set of spatial objects denoted as P, and a window w in a spatial constraint R, the *k*-CCPQ on windows refers to find the *k*-closest pairs of objects from P located in w.

For instance, consider the query: "find the four pairwise closest airports and hotels located in a specified city." For this problem, we set a street zone in this city as a query window, and expand it to find the required CPs.

Figure 1 gives an illustrative example of the *k*-CCPQ problem on windows, where the first data set is represented by stars while the second data set by crosses. Here, the window R depicted by the dashed line is a spatial constraint and the window w represented by the dashdotted line is an initial query window. In Fig. 1, we can observe that the 1-RCPQ is (p_2, q_1), and the 2-RCPQs are (p_2, q_1) and (p_2, q_2). Similarly, it is easy to find the 3-RCPQs, 4-RCPQs, etc.

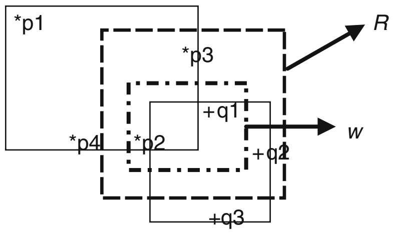

Fig. 1 Example of the *k*-CCPQs on window query problem

Some useful notations are given here. Let S and T be two spatial data sets, s be an index entry pointing to some node in a C-tree, *Node*(s) be the node that s points to, and *Sub_tree*(s) is the sub-tree rooted by *Node*(s). The C-tree stores the closest pair information by a triple (o_1,o_2, *dist*) along with the index entry pair (p_1, p_2). (o_1, o_2) is the closest pair of objects, where o_1 and $o_2$come from objects indexed by *Sub_tree*(p_1) and *Sub_tree*(p_2), respectively. In a C-tree, we borrow the buffer model proposed by Bhide [13] to manage the node update operations.

4.1 Node Insertion

C-tree uses the regular R-tree insertion algorithm to insert objects [7]. Let $\{(s_1, t_1), (s_2, t_2), \ldots, (s_n, t_n)\}$ be the index entry pairs pointing to the tree nodes that are along the insertion path, where s_1 points to the root node of S, s_n points to the leaf node that represents a newly inserted object, and *Node*(s_i) is a parent node of *Node*(s_{i+1}).

(s_i, t_i) needs to be updated iff there is an object r in the sub-tree rooted by *Node*(s_i), and an object r• in the sub-tree rooted by *Node*(t_i) satisfying dist(r, r•) < dist(s_i, t_i). If it finds such an object r•, then update the closest pair; otherwise, (s_i, t_i) keeps unchanged.

4.2 Node Update

If an object is changed, its index record must be deleted, updated, and then re-inserted. The node update algorithm is shown as follows:

Algorithm 1: Node Update

Input: two C-tree *A*, *B*
Output: two updated C-tree *A*', *B*'
Method:

```
 1. For (every object $p$ in $A$)
 2.   If (there is a path from the root node to $p$)
 3.     Update the node along the path;
 4.   End if
 5.   If (the object in $A$ does not have the node in the same level)
 6.     Update the leaf node in $B$;
 7.   End if
 8. End for
 9. For (every object $q$ in $B$)
10.   Apply the similar update operation as the object in $A$;
11. End for
```

In step 2, we update the node as finding the node in the same level in *B* by using the closest pair computation method of SRCP-tree [7]. Note, for the node deletion operation, if *p* is a leaf node, C-tree uses the plane-sweep algorithm [14] to find the new closest pair of objects.

4.3 Query Processing

In this section, we propose a new window query algorithm, namely WinQuery [6], to find the *k*-CCPQs, and borrow the idea of density-based *k*NN query algorithm [10] to this approach.

First, we use the EstiRange1 function [10] to approximate the query radius. The difference from EstiRange1 function is that GWCCP uses the square query method instead of the circle query used in [11]. The initial radius r_0 can be calculated by the following equation:

$$r_0 = \sqrt{\frac{k(x_u - x_l)(y_u - y_l)}{N}} \tag{4}$$

The query range *W* in the intersecting portion between the spatial constraint *R* and two existing spatial data sets *S* and *T*needs to be calculated first. The *x*-axis value of the upper-right corner of *W* is computed by Equation 5. Similarly, this equation can be applied to compute x_l, y_l, and y_u.

$$W.x_u = \min\{R.x_u, \max\{S.x_u, T.x_u\}\} \tag{5}$$

Second, we obtain the results from the estimated range by calling the WinQuery algorithm as shown in Algorithm 2 and the results are inserted into a temporary queue *temp*. θ, is a threshold that is used to determine whether to add a node or not, and its initial value is set as $\theta = \infty$, which represents a sufficiently large integer.

Then, create an empty priority query *priority* to store the closest pairs and find the *k*-CPQs within the new window. The algorithm terminates when *count* (the number of closest pairs found so far) is larger than or equal to *k*; otherwise, closest pairs have to be further obtained by gradually extending the window. The growing range is computed from the current window by the EstiRange2 function [10]. EstiRange2 returns a query radius denoted as r_n. The function used in this study is distinct from the one proposed in [10]. When $count = 0$, we use the factor of 1.5 (that is an empirical value by experiments) instead of 2 to expand the radius. When $count \in (0, k\text{–}1]$, the denominator under the radical sign does not have the factor of π, this is because we use square query. The query radius is defined in Equation 6.

Algorithm 2: WinQuery(C-tree *S*, C-tree *T*, Window *w*)

```
1. Create an empty queue temp;
2. If (both objects from S and T in the closest pair are inside w)
3.   Put the closest pair into a priority queue queue;
4. End if
5. While (queue is not empty)
6.   Pops one triple ($e_1$, $e_2$, dist) from queue;
7.   If (dist > $\theta$)
8.     Continue;
```

```
9.    End if
10.   If (both $Node(e_1)$ and $Node(e_2)$ are leaf nodes)
11.     For (every object $se_1 \in Node(e_1)$ and $se_2 \in Node(e_2)$ in $w$)
12.       If (the distance between these two objects is smaller than $\theta$)
13.         Update temp and $\theta$ = MinMinDist($se_1$, $se_2$);
14.       End if
15.     End for
16.   End if
17.   If ($Node(e_1)$ and $Node(e_2)$ are not leaf nodes)
18.     For (every object $se_1 \in Node(e_1)$ and $se_2 \in Node(e_2)$ in $w$)
19.       Prune such nodes $se_1$ and $se_2$ whose distances to their
        corresponding root nodes are greater than $T$;
20.       Compute MinMaxDist between $se_1$ and $se_2$ denoted as dist;
21.         If (dist < $\theta$ and there are $k$ elements in queue)
22.           Update $\theta$ to be the maximum value between the distance
        value of the top element in queue and dist;
23.           Compute the MinMinDist value between $se_1$ and $se_2$;
24.             If (it is less than $\theta$)
25.               Push ($se_1$, $se_2$, MinMinDist) to queue;
26.             End if
27.         End if
28.     End for
29. End if
30. If ($e_1$ is a leaf node and $e_2$ is an internal node)
31.     For(every object $se_2 \in Node(e_2)$ in $w$)
32.       Prune $se_2$ whose distance to its root node is greater than $\theta$;
33.     End for
34.   End if
35.   Use the similar manner as shown in lines 17–29 to handle the case
      that $e_1$ is a leaf node and $e_2$ is an internal node;
36. End while
37.   Pop all triples from temp;
```

$$r_n = \begin{cases} 1.5^* r_{n-1} & \text{if count} = 0 \\ \sqrt{\frac{k}{D(win)}} & \text{if } 0 < \text{count} \le \text{k}-1 \end{cases} \tag{6}$$

where $D(win)$ is the density of the window defined as follows:

$$D(win) = \frac{count}{(win.x_u - win.x_l) \times (win.y_u - win.y_l)} \tag{7}$$

Finally, we use the similar manner as shown in lines 5–36 to find other CPs.

The WinQuery algorithm plays an essential role in coping with *k*-CCPQ problem. The time complexity of WinQuery is similar to that of the Heap algorithm [4] and

SRCP-tree [7]. However, WinQuery performs better than the above two algorithms. This is because it uses a threshold θ as a filter to compress the size of the queue and applies an LRU buffer policy to cache the index entities. We will compare and analyze the performance of the above three algorithms in Section 5.

In WinQuery, we compute the closest pairs based on Equation (8). There is at least one pair (s_i, t_j) where $s_i \in S$, $t_j \in T$, such that:

$$\text{MinMinDist}(S,T) \leq \text{Dist}(s_i,t_j) \leq \text{MinMaxDist}(S,T) \tag{8}$$

Essentially, the proposed k-CCPQ algorithm can be generalized to handle other k-CPQ problems and is appropriate for other spatial objects. For instance, WindQuery is applicable to achieve continuous monitoring of nearest neighbors in highly dynamic scenarios where the objects move frequently and arbitrarily.

5 Experiments

The proposed algorithms were implemented in Java using spatial index library [15]. In order to measure the performance of GWCCP, we compare it with the typical k-CPQ processing algorithm, i.e., the Heap algorithm [4] that is non-recursive and evaluated to be better for the k-CPQ problem than other algorithms proposed in [4], and SRCP-tree [7]. To facilitate comparison, we extended Heap to handle the k-CCPQ problem in a given query range. Here, we call the new heap-based algorithm RHeap for short.

For each set of experiments, we use the following real-world and synthetic data sets.

- The synthetic data sets of distinct cardinalities are denoted by points with x and y coordinates. They are the sample data for the crime mapping and analysis tool CrimeStat [16] following a Bayesian distribution. They are used in the Journey to Crime module. CrimeStat is a spatial statistics package that can analyze crime incident location data and it provides a variety of tools for the spatial analysis of crime incidents [17].
- Two real-world data sets are from sample programs of Crime Travel Demand Module in CrimeStat. They consist of 65,536 traffic analysis zones represented by longitude and latitude. The data generating involves putting together the necessary data to estimate the model. This includes selecting an appropriate zone system, obtaining data on crime trips and allocating it to zones, obtaining zonal variables that will predict trips, creating possible policy, and obtaining one or more modeling networks [17]. The detailed description is available at http://www.icpsr.umich.edu/CRIMESTAT/files/CrimeStatChapter.11.pdf.

We conduct experiments on a PC of 2.4 GHz Pentium 4 processor with 512 MB of RAM. To comply with RHeap, the tree node capacity was set to 21, and the minimum capacity was set to 7, due to the reason given in [18]. To facilitate comparison, each experiment was run ten times and the average value was used to evaluate the performance.

We perform extensive experiments aiming to compare the performance of GWCCP with RHeap over two distinct data sets, and with SRCP-tree within the same data set, respectively, for the *k*-CCPQ problem in the crime databases. The experimentations consist of evaluating the effect of three important factors, i.e., the portion of overlapping between two data sets, the value of *k*, and the LRU buffer size.

5.1 Query Time Comparison of 1-CCPQ Algorithms

In this section, we compare the query time performance between GWCCP and RHeap. Both algorithms are evaluated with respect to the size of the data sets in distinct portions of overlapping between two data sets. To facilitate comparison, we assume zero buffer size for C-trees in this experiment.

Figure 2 illustrates the performance of both algorithms on 1-CCPQ between synthetic data under varying cardinality in (a) 0% and (b) 100% overlapping workspaces, respectively. The memory cost of 1-CCPQ for real-world data sets is shown in Fig. 3. Notice that the similar comparison results can be found for any other value of *k* as well, here we only give the results when *k*=1.

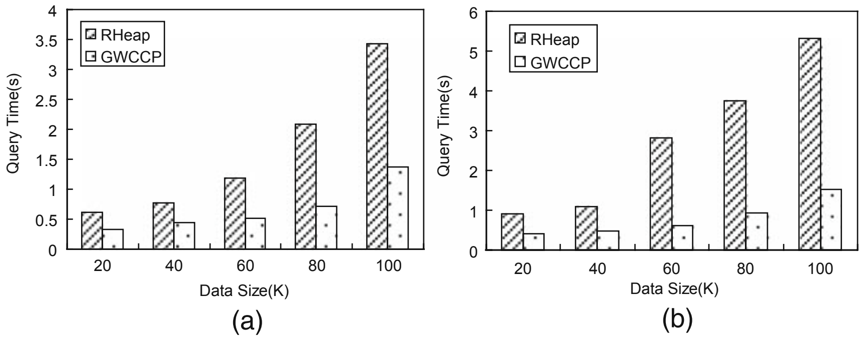

Fig. 2 Query time comparison of 1-CCPQ algorithms on synthetic data in: (**a**) 0% and (**b**) 100% overlapping workspaces

As shown in Fig. 2, GWCCP decreases the query time with respect to RHeap by a factor of 1.8–2.5 for no overlap cases between two data sets. When the data sets overlap (Fig. 2(b)), GWCCP also performs well, and achieves time saving of 1–4 times. One reason is that C-tree uses the shortest distance threshold θ instead of zero as the priority of the closest pairs, which increases the chance that a pair can be pruned from the priority queue in order to reduce the computations of *k*-CCPQs between two distinct data sets. In Fig. 3, the memory cost remains almost unchanged as the data set grows due to the threshold θ that helps save the memory. In addition, we can see that the query time in terms of GWCCP does not increase drastically with the data size.

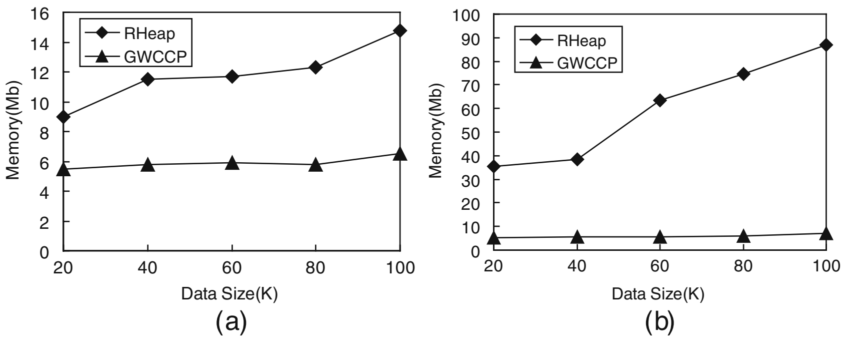

Fig. 3 Memory cost comparison of 1-CCPQ algorithms on real-world data in: (**a**) 0% and (**b**) 100% overlapping workspaces

We conclude that GWCCP is a better solution for the *k*-CCPQ problem, especially when the data sets significantly overlap. Since both algorithms are sensitive to the overlap factor, we have to further discuss the effect of this parameter in the following section.

5.2 The Effect of Overlap

We compare the memory cost of GWCCP with that of RHeap in the synthetic and the real-world data sets of 40 K cardinality as the overlapping percentage changes from 0 to 100%. The results are shown in Fig. 4.

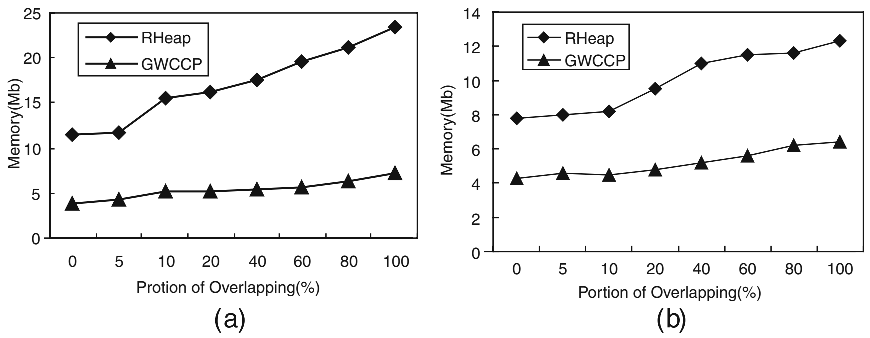

Fig. 4 Memory cost comparison on the overlap factor with: (**a**) synthetic and (**b**) real-world data

Straightforwardly, the overlap between two data sets plays a critical role for the performance. However, for GWCCP, the memory cost for a query involving larger overlap is slightly higher than the case involving disjointed workspaces. Because the query window extended gradually until finding the *k*-CPQs, and the growing query window helps eliminate the unneccessary memory cost spent on computing the distances between spatial objects. For RHeap, the higher the portion of overlapping between two data sets, the higher the memory cost. In summary, GWCCP is more suitable for *k*-CCPQ with a higher portion of overlapping between two distinct data sets.

5.3 The Effect of k

For this set of experiments, we run *k*-CCPQs in the real-world and in the synthetic data sets of 40 K cardinality, with *k* varying from 1 up to 100,000. Figures 5 and 6 illustrate the query time and the memory cost of both algorithms assuming (a) 0% and (b) 100% overlapping workspaces in the real-world data sets.

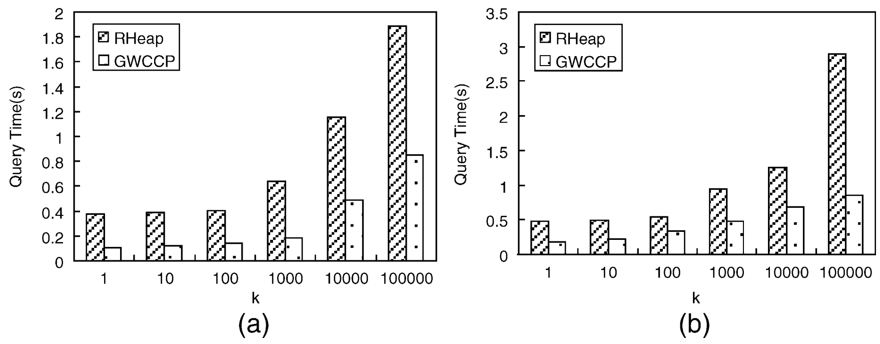

Fig. 5 Query time comparison of *k*-CCPQ algorithms with real-world data in: (**a**) 0% and (**b**) 100% overlapping workspaces

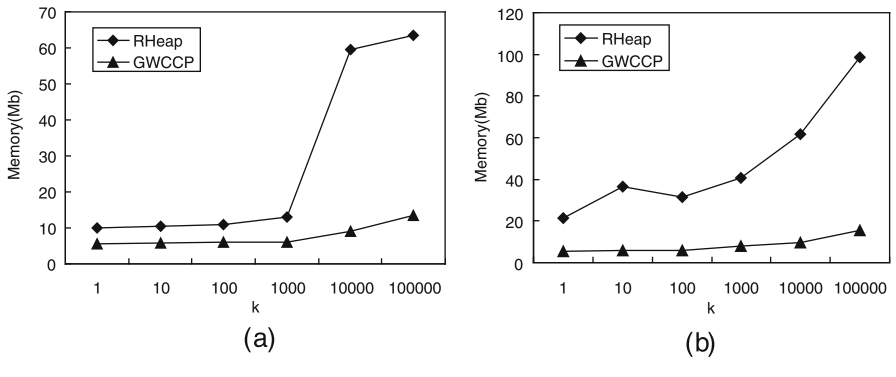

Fig. 6 Memory cost comparison of *k*-CCPQ algorithms with synthetic data in: (**a**) 0% and (**b**) 100% overlapping workspaces

According to Fig. 5, the cost of both algorithms increases with *k* in the real-world data sets. This is because both algorithms need more time to find the increased closest pairs of spatial objects as *k* grows. GWCCP wins RHeap with an average gap of 65% and 53% in 0% and 100% overlapping workspaces, respectively.

We can see from Fig. 6 that GWCCP achieves a significant improvement in terms of memory cost by a factor of 2–5 for 0% overlap (respectively, 4–7 for 100% overlap) as *k* grows. This further illustrates that GWCCP adapts to cope with the case of higher overlap between two data sets. According to Fig. 6, when the *k*value is larger than 1,000, the memory cost of RHeap changes sharply. However, GWCCP has a slight change when *k* increases. This is because C-tree uses the LRU buffer policy [13] to improve the quality of the C-tree update operation. In the following section, we will further investigate the effect of the LRU buffer.

5.4 Disk Access Comparison Under Distinct Buffer Size

Buffer policies considerably affect the performance of R-tree [19]. In this set of experiments, the buffer varies from $B = 0, \ldots, 256$ pages, i.e., each C-tree has equal portions of $B/2$ pages. We observe the performance of each algorithm in the real-world as well as the synthetic data sets of 40 K cardinality assuming 20% overlap and $k = 1000$. It is a tradeoff to choose a relatively low overlapping percentage, and RHeap performs well when*k* is lower than 1000 as empirically illustrated in the previous subsection. The results are shown in Fig. 7.

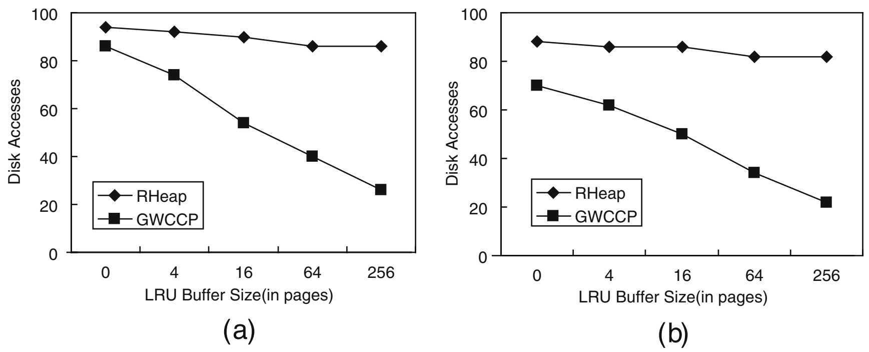

Fig. 7 Comparison of *k*-CCPQ algorithms under distinct LRU buffer size in: (**a**) synthetic and (**b**) real-world data sets

Figure 7 shows that the results for the synthetic and the real-world data sets are improved by up to 3 times in terms of disk accesses. However, RHeap is not sensitive to the buffer size (only up to 10% improvement as the buffer size reaches 256). On the contrary, GWCCP is sensitive to the buffer size. Because GWCCP uses the LRU buffer policy to maintain the C-trees, the disk accesses are greatly reduced.

5.5 Comparison Between GWCCP and SRCP-Tree

The problem of finding *k*-CCPQ in the same data sets is another practical problem in real-world scenarios. For example, a police officer may want to find the two closest first-aid centers. As suggested by [7], SRCP-tree performs well in handling this problem. In this section, we compare the performance in terms of the query time and memory cost of GWCCP with SRCP-tree.

In this set of experiments, we compare these two algorithms with an 80 K real-world and 80 K synthetic data sets. We first observe the query time of GWCCP and SRCP-tree as the overlapping percentage ranges from 0 to 100% with an interval of 20%. The results are shown in Fig. 8.

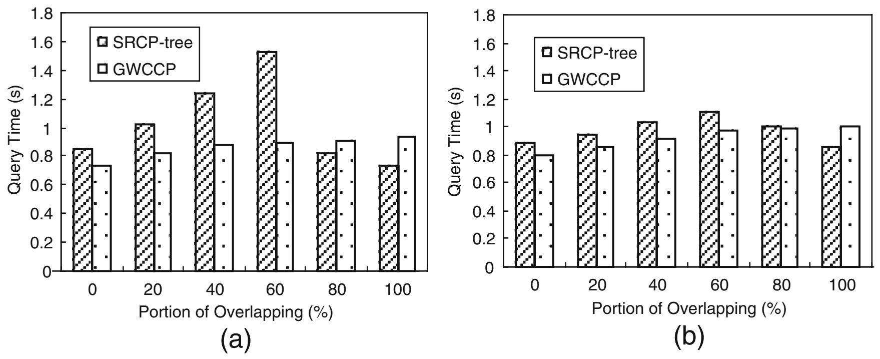

Fig. 8 Query time comparison within the same data set under distinct portions of overlapping on: (**a**) synthetic and (**b**) real-world data

As shown in Fig. 8, GWCCP outperforms SRCP-tree when the portion of overlapping is lower than 60% in the real-world and the synthetic data sets. For GWCCP, the query window is gradually extended until it finds the *k*-CCPQs (the overlapping percentage is about 60%), which helps eliminate the re-calculation of CPs. In particular, the increase of query time in terms of GWCCP is slight as it finds the *k*-CCPQs, whereas the query time in terms of SRCP-tree decreases drastically when the overlapping percentage is higher than 60%. This is because the probability of having a CP in the query range increases as it becomes large, and thus the query time drops.

We also compare the memory cost of these two methods within the synthetic and the real-world data sets, the results are shown in Fig. 9.

As we can see from Fig. 9, the memory cost of GWCCP keeps flat when the portion of overlapping reaches 60%, whereas the curve of SRCP-tree goes up linearly with the overlapping percentage. The reason is that GWCCP will stop searching when it finds the *k*-CCPQs, even if the portion of overlapping increases.

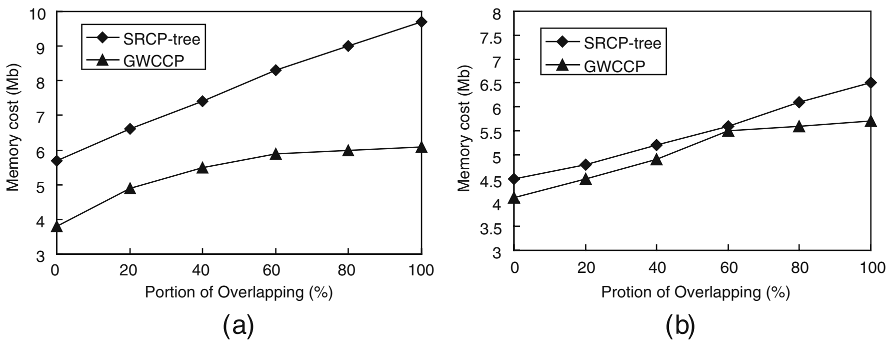

Fig. 9 Memory cost within the same data set under distinct portions of overlapping on: (**a**) synthetic and (**b**) real-world data

6 Conclusions and Future Work

We have proposed a new constrained *k*-closest pairs query processing algorithm based on growing windows, namely GWCCP. It employs an R-tree structure having inherent properties of the SRCP-tree to store the index entities, and a density-based range estimation method without boundary to calculate the square query range. Experiments have demonstrated that GWCCP outperforms the heap-based algorithm with respect to the portion of overlapping between two distinct data sets, the value of *k*, and the LRU buffer size.

We point out several interesting issues for future research including developing other range estimation approaches for *k*-CCPQs, proposing other good buffer policy for C-trees, and applying our proposed algorithm to assist in crime mapping and data analysis.

Acknowledgments This work is supported by the National Natural Science Foundation of China under Grant No. 60773169, the 11th Five Years Key Programs for Sci. and Tech. Development of China under Grant No. 2006BAI05A01, and the Youth Software Innovation Project of Sichuan Province under Grant Nos. 2007AA0032 and 2007AA0028.

References

1. Overview of Data Collection in British Columbia. Available at http://www.pssg.gov.bc.ca/police_services/publications/
2. Chen H (2007) Exploring extremism and terrorism on the Web: the dark web project. In: Pacific Asia Workshop on Intelligence and Security Informatics, PAISI 2007, Chengdu, pp. 1–20
3. Leipnik MR, Albert DP (2003) GIS in law enforcement: Implementation issues and case studies. Routledge, London, pp. 1–47
4. Corral A, Manolopoulos Y, Theodoridis Y, Vassilakopoulos M (2000) Closest pair queries in spatial databases. In: Proceedings of ACM SIGMOD 2000, Dallas, pp. 189–200

5. Corral A, Manolopoulos Y, Theodoridis Y, Vassilakopoulos M (2004) Algorithms for processing *k*-closest pair queries in spatial databases. Data and Knowledge Engineering, 49(1): 67–104
6. Qiao S, Tang C, Jin H, Dai S, Chen X (2008) Constrained *k*-Closest Pairs Query Processing Based on Growing Window in Crime Databases. In: 2008 IEEE International Conference on Intelligence and Security Informatics, Taipei, pp. 58–63
7. Shan J, Zhang D, Salzberg B (2003) On spatial-range closest-pair query. In: Proceedings of SSTD 2003, Greece, pp. 252–270
8. Guttman A (1984) R-trees a dynamic index structure for spatial searching. In: Proceedings of ACM SIGMOD 1984, Boston, pp. 47–57
9. Roussopoulos N, Kelley S, Vincent F (1995) Nearest neighbor queries. In: Proceedings of ACM SIGMOD 1995, San Jose, pp. 71–79
10. Liu D, Lim E, Ng W (2002) Efficient k nearest neighbor queries on remote spatial databases using range estimation. In: Proceedings of SSDBM 2002, Edinburgh, pp. 121–130
11. Hjaltason GR, Samet H (1998) Incremental distance join algorithms for spatial databases. In: Proceedings of ACM SIGMOD 1998, Seattle, pp. 237–248
12. Ferhatosmanoglu H, Stanoi I, Agrawal D, El Abbadi A (2001) Constrained Nearest Neighbor Queries. In: Proceedings of SSTD 2001, Redondo Beach, pp. 257–278
13. Bhide A, Dan A, Dias D (1993) A simple analysis of LRU buffer replacement policy and its relationship to buffer warm-up transient. In: Proceedings of ICDE 1993, Vienna, pp. 125–133
14. Hsiao P, Tsai C (1990) A new plane-sweep algorithm based on spatial data structure for overlapped rectangles in 2-D plane. In: COMPSAC'90, Chicago, pp. 347–352
15. http://research.att.com/˜marioh/spatialindex/index.html
16. http://www.icpsr.umich.edu/CRIMESTAT/
17. Ned Levine (2007) CrimeStat: A Spatial Statistics Program for the Analysis of Crime Incident Locations (v 3.1). Ned Levine and Associates, Houston, TX, and the National Institute of Justice, Washington, DC. March.
18. Beckmann N, Kriegel HP, Schneider R, Seeger B (1990) The R$*$-tree: an efficient and robust access method for points and rectangles. In: Proceedings of ACM SIGMOD 1990, New York, pp. 322–331.
19. Leutenegger ST, Lopez MA (2000) The effect of buffering on the performance of R-trees. IEEE Transactions on Knowledge and Data Engineering, 12(1): 33–44.