On Spatial-Range Closest-Pair Query

Jing Shan, Donghui Zhang, and Betty Salzberg*

{jshan,donghui,salzberg}@ccs.neu.edu
College of Computer and Information Science
Boston, MA 02115

Abstract. An important query for spatial database research is to find the closest pair of objects in a given space. Existing work assumes two objects of the closest pair come from two different data sets indexed by R-trees. The closest pair in the whole space will be found via an optimzed R-tree join technique. However, this technique doesn't perform well when the two data sets are identical. And it doesn't work when the search range is some area other than the whole space. In this paper, we address the closest pair problem within the same data set. Further more, we propose a practical extension to the closest pair problem to involve a query range. The problem now becomes finding the closest pair of objects among those inside a given range. After extending the existing techniques to solve the new problem, we proposed two index structures based on augmenting the R-tree and we also give algorithms for maintaining these structrures. Experimental results show that our structures are more robust than earlier approaches.

1 Introduction

Spatial databases have received more and more attention recently. Some major database vendors have provided spatial database support, e.g. Oracle Spatial Cartridge [1] and IBM Informix Spatial DataBlade [2]. Examples of spatial database applications include mapping, urban planning, transportation planning, resource management, geomarketing, archeology and environmental modelling [6].

One important spatial database query is the *closest pair (CP)* query, which is to find the closest pair of objects among two data sets. One may be interested in finding the closest middle school and bar, the closest supermarket and apartment building, etc. This topic has good solutions in the computational geometry area [17, 7]. However, only recently has this problem received interest in the environment of spatial databases [10, 5, 6, 21]. All of these focus on two disjoint data sets. In fact, Corral et al. have studied the effect of overlapping between two data sets and found out that "a small increase in the overlap between the data sets may cause performance deterioration of orders of magnitude" [5, 6]. Obviously, when two data sets are identical, there is extensive overlapping. Although in their technical report [5], Corral et al. introduced a way to extend

* This work was partially supported by NSF grant IIS-0073063.

their algorithms to solve the case when two data sets are the same, they also pointed out that "this case still deserves careful attention".

Furthermore, to the best of our knowledge, none of the existing work has addressed the *Range-CP* problem, i.e. *Given a spatial range R, find the closest pair of objects located within R.* For example, one may be interested in finding, within Massachusetts, the closest (high school, bar) pair. The CP problem is a special case of the Range-CP problem when the query range is the whole space. The Range-CP problem is more interesting for the following reason. In the original CP problem, the pair of closest objects is fixed. Thus we can perform the query once and store the answer and we need only perform the query again after there are inserts or deletes of objects. The Range-CP problem, however, is more difficult. The query range can be arbitrary. To maintain a closest pair of objects for every possible spatial range is not practical.

The contributions of the paper are as follows:

- We propose a practical extension to the Closest Pair problem to involve a query range. We address this new Range-CP problem in the case when the two data sets are identical. An example query is: "what are the closest pair of post offices in Boston?" If the City Hall wants to move some post office to a new location, this query result may help decision making. The query also applies to finding closest pairs of different types of objects, e.g. between high schools and bars, when both are indexed in the same R-tree.
- We discuss how to extend existing CP solutions to solve the Self Range-CP problem.
- We propose two versions of the SRCP-tree, a new index structure, to solve this problem.
- We identify a practical issue not addressed in the most recent solution for CP problem [21] and provide solutions, thus improving the algorithm.
- Experimental results are provided which demonstrate the efficiency of our approaches.

The rest of the paper has the following organization: Section 2 formally defines the problem to be addressed and provides related work. Section 3 presents our extension to existing techniques to solve the new problem. Our new solutions appear in section 4. We provide experimental results in section 5. Finally, section 6 contains our conclusions and future directions.

2 Problem Definition and Related Work

2.1 Problem Definition

The CP query finds a pair of objects from two data sets with minimum distance between them. If there are more than one pairs with this distance. The CP query returns one of them. The formal definition is given below:

Definition 1. *Given two spatial data sets S and T, the* **Closest Pair (CP) query** *finds the pair of objects (s, t) such that $s \in S$, $t \in T$, and $\forall s' \in S$ and $t' \in T$, $distance(s,t) \leq distance(s',t')$.*

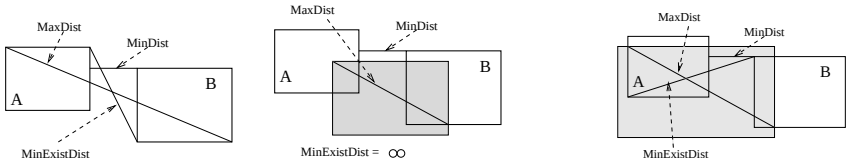
Here $distance(s,t)$ corresponds to the Euclidean distance of two objects s and t. A practical variation which we do not see in the literature is the *Range-CP* problem, where a spatial range R is involved and we are interested in finding the closest pair of objects inside R. The SRCP query finds the closest pair of objects from a subset of objects in one data set, where the subset of objects are those whose locations are in a given spatial range. More formally,

Definition 2. *Given a spatial data set S, the* **Self Range Closest Pair (SRCP) query** *regarding to a spatial range R finds the pair of objects (s, t) such that $s,t \in S$, $s \neq t$, R contains s and t, and $\forall s', t' \in S$ such that $s' \neq t'$ and R contains s' and t', $distance(s,t) \leq distance(s',t')$.*

Both these problems have a variation of finding k closest pairs instead of only one. Here $k \geq 1$ is given at run time.

2.2 The R-tree and Some Useful Metrics

Spatial database index structures have received a lot of attention from researchers. Various indexing techniques have been proposed, e.g. the R-tree family [9, 3]. An review of spatial index structures appears in [8]. In an R-tree, every node has an *minimum bounding rectangle (MBR)* and the MBR of a higher-level tree node contains the MBRs of all its children. Thus when performing a range query, we only need to browse the sub-trees whose root-level MBRs intersect with the query range.



(a) without a query range (b) with a query range (c) another example with a query range

Fig. 1. Two MBRs and their *MinDist*, *MaxDist* and *MinExistDist*.

Given two MBRs A, B of R-tree nodes, following [15], we define some useful metrics. $MinDist(A,B)$ is the smallest distance between A and B boundaries. It is a lower bound of the distance between an object enclosed in A and an object enclosed in B. Similarly, $MaxDist(A,B)$ is the largest distance between A and B boundaries and it is a upper bound of the distance between an object enclosed in A and an object enclosed in B. Another metric is $MinExistDist(A,B)$, which is the minimum distance which guarantees that there exists a pair of objects, one in A and the other in B, with distance closer than the metric. In other words, $MinExistDist$ expresses an upper bound for the solution of the closest pair problem, since the closest pair in the collection will be closer or equal in

distance to that of any $MinExistDist$ for any pair A, B of R-tree nodes. An example illustrating these metrics is shown in Figure 1a, which is borrowed from [6]. To summarize, the following hold:

$$\forall o_1 \in A, o_2 \in B, MinDist(A, B) \leq dist(o_1, o_2) \leq MaxDist(A, B)$$

$$\exists o_1 \in A, o_2 \in B, dist(o_1, o_2) \leq MinExistDist(A, B)$$

The papers [6, 15], although using these metrics, did not consider query ranges. We extend these metrics to the case when there is a query range R involved, as shown in the shadowed rectangle in Figure 1b and 1c. As we can see in both figures, the value of $MinDist$ remains the same, and the value of $MaxDist$ changes. We can prove that $MaxDist(A, B, R) = MaxDist(A \cap R, B \cap R)$. Intuitively, the $MaxDist$ with a query range R is equal to the $MaxDist$ of the following MBRs without any query range: the intersection of A with R, and the intersection of B with R.

The case of $MinExistDist(A, B, R)$ is tricky. It can be $+\infty$ as shown in Figure 1b. This is because it is possible that there is no object from at least one MBR that is in this range. In this case, however large a distance is, we cannot guarantee that there exists a pair of objects in this range. However, in Figure 1c, $MinExistDist$ is not ∞. The reason is that the query range contains (at least) one edge from each MBR.

2.3 Related Work

The closest pair query has been studied in computational geometry [17, 7]. More recently, this problem has been approached in the environment of spatial databases [10, 5, 6, 21]. As far as we know, Hjaltason and Samet [10] were the first to address this issue. Their paper proposed an incremental join algorithm between two R-tree indices. The idea is to maintain a priority queue which contains pairs of index entries and objects, and pop out the closest pair and process it.

Later, Corral et al. [6] proposed an improved version known as the *heap algorithm*. One improvement, which was also proposed simultaneously by Zhang et al. [23, 24], is the *virtual height optimization* which can be applied when joining two R-trees with different height. A benefit of this optimization is that we can avoid pushing a pair of objects (from the shorter tree) and internal node (from the higher tree) into the queue. Another improvement is that there is no need to push any pair of objects into the queue at all. Rather, we remember a threshold value T as the distance of the closest pair of objects we have seen. Instead of pushing a pair of objects into the queue, we use it to update T if the distance is smaller than T.

The most important optimization is that we can use the value T as a filter to shrink the size of the queue dynamically. If the $MinDist$ of a pair of nodes is no smaller than T, there is no way the closest pair of objects can be found by processing the pair of nodes, and thus there is no need to maintain it in the queue. Another important optimization is that if $MinExistDist$ of a pair of

nodes is smaller than T, we can reduce T without actually seeing the pair of objects with distance equal to or smaller than T.

However, both the above algorithms are not efficient when the two R-trees have extensive overlapping. In the case when two nodes intersect, their $MinDist$ is 0. Both algorithms need to process all pairs of intersecting nodes. If the two R-trees are identical, since the distance between every page and itself is 0, the query algorithm has to at least check every tree page.

Recently, Yang and Lin [21] proposed a new structure called the *b-Rdnn tree* which has a better solution to the CP problem when there is overlap between the two data sets. The idea is to find k objects from each data set which are the closest to the other data set. Here the distance between an object and a data set is the minimum distance between the object and all objects in the data set. The paper proves that the closest pairs can be found by examining these $2k$ objects. In order to efficiently find these $2k$ objects, along with each object, the distance *dnn* to its nearest neighbor in the other data set is maintained. Also, along with each index entry, a value *min_dnn* (*max_dnn*) is maintained which is the minimum (maximum) *dnn* value of all objects in the sub-tree.

However, this approach does not apply to our new query where there is a query range involved. The thing is, the object with the smallest distance to the data set may not be inside the query range. Even if it is, its nearest neighbor of it may not be in the query range.

Also related are papers [15, 4, 12, 16, 11] on nearest neighbor queries and papers [22, 14, 19] on reverse nearest neighbor queries. In addition, there are nearest neighbor algorithms for moving object databases [13, 18].

2.4 Improvement on An Existing CP Problem Solution

In this section, we identify a very practical issue not addressed in the most recent solution to the CP problem proposed by Yang and Ling [21]. The paper contains a theorem: "for any pair of objects (o_1, o_2), which is one of the k closest pairs between S and T, $o_1 \in kNN(S,T)$ and $o_2 \in kNN(T,S)$". Here $kNN(S,T)$ is the k objects from S that are the closest to T.

We argue that the theorem and the algorithm based on it are valid only when no two pairs of objects have the same distance. In a practical scenario, it is quite possible that different pairs of objects may have the same distance. If so, the algorithm there may not be correct. Figure 2 shows an example. Objects s_1, s_2 belong to set S, while objects t_1, t_2 belong to set T. Assume $Dist(s1,t1) = Dist(s2,t2)$ is the minimum distance between any object in S and any object in T. Consider the case when k=1, i.e. we want to find only one closest pair between S and T. It is possible that the algorithm uses the bRdnn-tree on S to find s_1 (since the distance between s_1 and T is minimal), while it uses the bRdnn-tree on T to find t_2. Now, the only candidate for the closest pair is (s_1, t_2), which leads to an error.

We propose two solutions to this problem. One solution works as follows. In order to find the k closest pairs, instead of finding k objects from S which are the closest to T (as in [21]), we find also all objects whose distance to T is equal

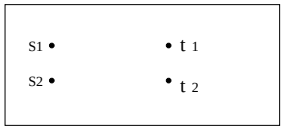


Fig. 2. When different pairs of objects can have the same distance, the algorithm of [21] may not be correct.

to the distance between the k^{th} object and T. Similarly, we find more than k objects from T which are the closest to S. Let's say we find $k+x$ objects from S and we find $k+y$ objects from T. These two sets of objects form a set of $(k+x)(k+y)$ pairs, and we can prove that the k closest pairs of objects must belong to this set.

For the other solution, we only find k objects from each data set which are the closest to the other set. They form a set of k^2 pairs. However, along with each object thus found, we also find its nearest neighbor in the other data set. If we union the k^2 pairs with the $2k$ NN pairs, we get a set of k^2+2k pairs. We can prove that the k closest pairs of objects must come from this set.

3 Straightforward Solutions

In this section, we extend existing solutions to the CP problem to solve the Self Range Closest Pair (SRCP) problem.

3.1 Range Query Followed by Plane Sweep

We know that spatial access methods like the R-tree support range queries. We also know that when all objects are in memory, there are computational geometry solutions to the CP problem. So one straightforward approach, assuming the range query result is not too big, is to perform a range query on the data set and then perform some computational geometry method, e.g. the well-know *sweeping* technique, on the range query result.

To find the closest pair, the sweeping technique works as follows. Read objects in increasing x order, while maintaining the closest pair found so far. Whenever an object is read into the buffer, compare it with all objects that are already in the buffer to determine if the distance is smaller than the currently maintained closest pair. If no object is ever removed from the buffer, this is obviously an $O(n^2)$ algorithm. However, objects which can be determined not to be close to any non-read objects can be dynamically removed from the buffer. Here we use the distance of the currently maintained closest pair as a threshold, and any object whose distance in the x dimension to the currently sweep line is larger than this threshold, can be removed from the buffer.

3.2 Extending the Incremental Join Algorithm and Heap Algorithm

Both the incremental join algorithm of [10] and the heap algorithm of [6] can be extended to solve the SRCP problem. To extend to the Self-CP case, we just assume there are two copies of the same data set and apply the original algorithm to the two copies. A minor modification is that the pair of an object and itself should not be counted as a closest pair.

The extension to the Range-CP case is as follows. Whenever a pair of internal nodes are about to be pushed into the queue, we check to make sure that both nodes intersect with the query range. Whenever a pair of objects are about to be compared, we check to make sure that both of them are inside the query range. Also, an important thing is that we have extended the definition of $MinExistDist$ to include a query range, as discussed in section 2.2.

4 Our solutions

There are two motivations to our solutions to the SRCP problem.

- Suppose we maintain the closest pair of objects in a system table outside the R-tree used for the general SRCP problem. Upon a query where $k = 1$, if it turns out that both these two objects fall in the query range, we can immediately determine that this pair is the SRCP query result without accessing any tree node. None of the existing solutions in section 3 has this ability.
- When applying the incremental join algorithm or the heap algorithm (section 3.2) to the SRCP problem, the distance of every "self-pair" of nodes is zero and thus every self-pair of tree nodes intersecting the query range must be pushed into the processing queue and processed. However, if along with each index node, we store the shortest distance of objects in the sub-tree, we can then use this value instead of zero as the priority of the self-pair. This increases the chance that a pair can be eliminated from the queue and improves performance.

Based on these two motivations, we develop two versions of the SRCP-tree, which is presented in section 4.1 and 4.2. We then analytically compare them with the straightforward approaches in section 4.3. For simplicity, we focus on finding a single closest pair, and we discuss how to extend to finding k closest pairs in section 4.4.

4.1 SRCP-tree (Version one)

In this section, we define the first version of the SRCP tree. We first define some notation. Let e be an index entry pointing to some node in the R-tree. We use $Node(e)$ to represent the node that e points to. We use $Subtree(e)$ to represent the sub-tree rooted by $Node(e)$.

Tree Structure The SRCP-tree (version one) is an R-tree augmented with "local" closest pair information. Each index entry e of the R-tree is augmented with a triple (o_1, o_2, $dist$). Here o_1 and o_2 are the closest pair of objects in the sub-tree rooted by $Node(e)$. The value $dist$ is the distance between o_1 and o_2. (Since it can always be computed from o_1 and o_2, we could choose not to store $dist$.) In a small auxiliary tree descriptor, we store the closest pair of objects in the whole tree. We can think of this pair as being associated with the entry pointing to the root node of the tree.

Closest Pair Computation Given an R-tree, we discuss here how to compute the augmented closest pair information to be stored along with an entry e.

Case 1: $Node(e)$ is a leaf page. We can use a plane-sweep algorithm to compute the closest pair of objects among the objects in $Node(e)$.

Case 2: $Node(e)$ is an index page. Suppose the index entries stored in $Node(e)$ are $e_1, \ldots, e_t$, and the augmented information for these entries is computed already. Our task is to find the closest pair (p, q) in the sub-tree rooted by $Node(e)$. Without loss of generality, assume among pairs of objects stored in $e_1, \ldots, e_t$, the pair in e_i has the smallest distance. We know that $distance(p, q) \leq e_i.dist$, and that if $distance(p, q) \neq e_i.dist$, p and q are in different sub-trees. So, we can perform an incremental join on every pair of (e_x, e_y) such that $MinDist(e_x, e_y) < e_i.dist$. Note that here we use a threshold $T = e_i.dist$ to determine whether we join a pair of nodes or not. In fact, the threshold T can be dynamically reduced by: (a) when joining e_x and e_y, if $MinExistDist(e_x, e_y) < T$, set $T = MinExistDist(e_x, e_y)$; and (b) when a pair of objects is found with a smaller distance than T, assign that distance to T.

Insertion Suppose we want to insert a new object o. First, we follow the regular R-tree insertion algorithm to insert the object. We can prove that, among all entries stored in the tree, only those pointing to tree nodes along the insertion path may need to be updated. Let $e_1, \ldots, e_h$ be the index entries pointing to the tree nodes that are along the insertion path. Here e_1 points to the root node, e_h points to the leaf node where the new object o was inserted, and $Node(e_i)$ is a parent node of $Node(e_{i+1})$.

The closest pair stored at e_i needs to be updated if and only if there exists an object o' in the sub-tree rooted by $Node(e_i)$ such that $Distance(o, o') < e_i.dist$. So, we can perform a range search on $Subtree(e_i)$ to see if there exists any object whose distance to o is less than $e_i.dist$. If we find such an object o', we update the closest pair stored at e_i as (o, o'). Otherwise, e_i does not need to be updated.

Notice that these range searches have overlap. For instance, when determining the new closest pair for e_i, we may check $Subtree(e_{i+1})$, since it is part of $Subtree(e_i)$. However, when determining the new closest pair for e_{i+1}, we may check $Subtree(e_{i+1})$ again. These searches can be combined in the following algorithm: (a) Perform a range search on $Subtree(e_1)$ with radius $e_1.dist$ around the new object o, ignoring $Subtree(e_2)$. If any object is found, update the closest pair stored at e_1. (b) Perform a range search on $Subtree(e_2)$ with radius

$e_2.dist$ around o, ignoring $Subtree(e_3)$. If any object o' is found, update e_2. If $distance(o, o') < e_1.dist$, update e_1 as well. Extending this method to lower levels of the tree is straightforward.

Now we discuss node splitting. If a node N is split into two, N and N', we need to update the local closest pair information stored at the entries pointing to the two nodes. This can be done by the close-pair computation algorithm given above.

Deletion Similar to the insertion case, after we delete an object o using the R-tree deletion algorithm, in order to maintain the augmented information, only the entries pointing to nodes on the deletion path need to be updated. Let e_i be an entry on the deletion path. The closest pair stored at e_i needs to be updated if and only if the closest pair stored at e_i contains o. Further, we differentiate two cases.

Case 1: $Node(e_i)$ is a leaf node. We perform a plane-sweep algorithm to find the new closest pair of objects in $Node(e_i)$.

Case 2: $Node(e_i)$ is an index node. We perform the closest pair computation algorithm above. But before we do that, there is an optimization. Suppose there exists an index entry se stored in $Node(e_i)$ where $se.dist = e_i.dist$, and the closest pair stored at se does not contain o. In this case, we can simply copy the closest pair stored at se to e_i.

If two tree nodes merge into one, we update the closest pair stored at the new node by performing the closest pair computation algorithm.

Query The query algorithm for the SRCP-tree is given below.

Algorithm *SRCPQuery*(SRCP-tree S, Range R): Given a SRCP-tree S and a spatial range R, find the closest pair of objects of S that are inside R.

1. If both $S.o_1$ and $S.o_2$ are inside R, return $(S.o_1, S.o_2)$.
2. Push (S, S, $S.dist$) into a priority queue Q, ordered by the third value of each triple.
3. Initialize threshold $T = \infty$ and the query result $(res_1, res_2) = NULL$.
4. **while** (Q is not empty)
   (a) Pop the triple $(e_1, e_2, dist)$ from Q which has the smallest $dist$ value.
   (b) **if** ($Node(e_1)$ is a leaf node)
   for (every object $o_1 \in Node(e_1)$ and $o_2 \in Node(e_2)$ that are inside R and $o_1 \neq o_2$)
   If $distance(o_1, o_2) < T$, update $(res_1, res_2) = (o_1, o_2)$, $T = distance(o_1, o_2)$, then optimize Q.
   end for
   Goto step 4.
   end if
   (c) /* $Node(e_1)$ is an index node */
   for (every entry $se_1 \in Node(e_1)$ and $se_2 \in Node(e_2)$ that both intersect R)
   i. **if** ($se_1 = se_2$)

A. If $se_1.dist \geq T$, continue the for loop.
B. If both $se_1.o_1$ and $se_1.o2$ are inside R, update $(res_1, res_2) = (se_1.o_1, se_1.o_2)$, $T = se_1.dist$, then optimize Q, and then continue the for loop.
C. Push $(se_1, se_1, se_1.dist)$ into Q.

ii. **else** /* $se_1 \neq se_2$ */
A. If $MinExistDist(se_1, se_2, R) < T$, update $T = MinExistDist(se_1, se_2, R)$ and optimize Q;
B. If $MinDist(se_1, se_2) < T$, push $(se_1, se_2, MinDist(se_1, se_2))$ into Q;
end if
end for
end while

5. return (res_1, res_2);

We explain the algorithm in a little detail. Let S denote both the SRCP-tree and the index entry pointing to the root node of the SRCP-tree S. Since the closest pair of objects (without a range) is stored along with S, step 1 of the algorithm checks to see if both objects in the closest pair are inside the query range R. If yes, we have already found the result and the algorithm finishes. Otherwise, we initialize the processing queue Q by pushing the pair of root entries into Q.

Each element in Q contains a pair of index entries and a distance value. For a pair of two different entries, the value is their $MinDist$. For a pair of the same entry, the value is the closest-pair distance stored in the entry. This distance is a lower bound of the distance of any pair of objects to be found by examining the pair. Thus if during execution of the algorithm, a pair of objects is found with distance smaller than or equal to this distance of some triple in Q, the triple can be safely erased from Q. In fact, this is what we mean when we say "optimize Q" later in the algorithm.

The major task is done in step 4, which pops from Q one triple at a time and processes it. Let the triple be (e_1, e_2, $dist$). Step 4(b) handles the case when e_1 points to a leaf node (e_2 points to a leaf node as well). We examine every pair of objects, one from each node, that are inside R. If the distance between the two objects is smaller than the currently maintained threshold T, we remember this pair and update T. Note that at this step, we also optimize Q as follows: we remove all triples from Q where $dist$ is no smaller than T.

Step 4(c) handles the case when e_1 and e_2 point to index nodes. In this case, we check every child entry of $Node(e_1)$ against every child entry of $Node(e_2)$. Of course, we only consider the child entries that intersect the query range R. Let the pair of child entries be (se_1, se_2).

A subcase, as shown in step 4(c)i, is when se_1 and se_2 are actually the same entry. Existing algorithms always push such self-pairs into Q, since the distance of a self-pair is regarded to be zero. As shown in step 4(c)iA, our algorithm may avoid pushing this pair into Q, since we already know the minimum distance of any two objects in the sub-tree. Even if the distance of the closest pair stored

along with se_1 is smaller than the current threshold T, it is still possible that our algorithm avoids pushing the pair into Q. This case is shown in step 4(c)iB, when both objects in the closest pair stored at se_1 are inside R.

Another subcase, as shown in step 4(c)ii, is when entries se_1 and se_2 point to different nodes. This case is the same as the heap algorithm. First, we check to see if their $MinExistDist$ is smaller than threshold T. If yes, since it is guaranteed that by joining this pair, we can find a pair of objects with distance no larger than that, we can go ahead and update threshold T. After that, we compare the $MinDist$ of the two entries with T and we can avoid pushing this pair into Q if the $MinDist$ is no smaller than T.

4.2 SRCP-tree (Version Two)

Tree structure Instead of storing "local" closest pair information $(o_1, o_2, dist)$ in each index entry e as discussed in the previous section, in version two of our structure we store *"local-parent"* closest pair information (o_c, o_p, $dist$) along with each index entry e. If $Node(e)$ is not the tree root, (o_c, o_p) is the closest pair of objects where o_c comes from objects indexed by $Subtree(e)$ and o_p comes from objects indexed by the subtree rooted by the parent of $Node(e)$. If $Node(e)$ is the tree root, (o_c, o_p) is the closest pair of objects for the whole tree. In both cases, $dist$ is the distance of the two corresponding objects and we could choose not to store it but compute it when in need.

Insertion The insertion algorithm is similar to that of version one. Besides the R-tree insertion algorithm, we need to update the local-parent closest-pair information. We start with the entry S pointing to the root node. Since at the root entry, both versions of the SRCP-tree maintain the same information: the closest pair for the whole tree, to update the information in the second version is the same as in version one.

Now let's go one level down. Suppose the child entries stored inside of $Node(S)$ are A, B and C. Assume the new object o is inserted into $Subtree(B)$. We need to update B if there exists an object in $Subtree(S)$ whose distance to o is smaller than $B.dist$. We can find such an object by performing a range search on $Subtree(S)$. The insertion of o may also affect the information stored at A, if the distance between o and some object in $Subtree(A)$ is smaller than $A.dist$. In order to find such an object, we perform a range search on $Subtree(A)$. The case for C is similar. Notice that this does not mean the whole tree needs to be updated. For instance, the information stored at the children of $Node(A)$ and the children of $Node(C)$ do not need to be updated. Basically, we check the insertion path, and for each entry along the insertion path, we check its sibling entries as well.

Deletion

The deletion algorithm needs to be modified from version one of the structure accordingly. Basically, for each index entry e along the deletion path, we need to modify the information stored at e and the sibling entries of e.

Query The query algorithm here is similar to that of the first version. The difference lies in step 4(c), i.e. when a pair of entries (e_1, e_2) pointing to index nodes are popped from Q and are to be processed. We differentiate two subcases. Case one is when e_1 and e_2 are different entries. In this case, we process every pair of child entries, one from $Node(e_1)$ and the other from $Node(e_2)$, in the same way as step 4(c)ii of the version-one query algorithm does. In the rest, we focus on the other subcase, when an entry e is to be joined with itself.

Let A, B be sibling index entries. If $A.dist \leq B.dist$ and both objects in the closest pair stored at A are inside the query range R, we can determine that there is no need to process any pair consisting of B and a sibling. The reason is that the minimum distance of an object in $Subtree(B)$ and an object in the sub-tree rooted by any of its siblings (including B itself) is $B.dist$, which is no better than the pair stored with A, which we already found.

The above observation motivates our algorithm. To process the pair (e, e), we first determine, among all child entries in $Node(e)$ where both objects in the maintained closest pair are inside R, which one has the smallest $dist$. This smallest distance is defined as $\boldsymbol{SmallestExistDist}(e, R)$. If this smallest distance is smaller than the current threshold T, we can safely update T and update the currently found closest pair, then optimize Q.

Next, we consider another concept: $\boldsymbol{SmallestDist}(e) = min\{se.dist | se \textit{ is a child entry in } Node(e)\}$. This is the smallest $dist$ among those maintained along with every child entry in $Node(e)$, independent to any query range. Obviously, we have: $SmallestDist(e) \leq SmallestExistDist(e, R)$. If the two values are equal, there is no need to push any pair of child entries of e to the processing queue Q at all. This is because, by examining sub-trees of $Node(e)$, the best we can get is a pair of objects with distance being $SmallestDist(e)$, and we have already found such an object with distance $SmallestExistDist(e, R)$.

Now, the only case left is when $SmallestDist(e) < SmallestExistDist(e, R)$. In this case, we examine every pair of child entries (se_1, se_2) of $Node(e)$ that intersect range R. We push this pair into Q if all the following three conditions holds: (a) $MinDist(se_1, se_2) < T$, (b) $se_1.dist < SmallestExistDist(e, R)$, and (c) $se_2.dist < SmallestExistDist(e, R)$. In this case, we push (se_1, se_2, $max\{se_1.dist, se_2.dist\}$) into Q.

4.3 Analytical Comparison of The Algorithms

Our SRCP-trees are obviously better than the straightforward approach of performing a range query and then finding the closest pair on the query result, especially if the query range is large. When there are many objects in the range query result, to maintain them in memory and process them is costly.

Our SRCP-trees should have better query performance than both the incremental join algorithm [10] and the heap algorithm [6]. The main reason is that the two algorithms join every intersecting pair. In particular, in our addressed problem, the two data sets being joined are identical, which leads to extensive overlapping. For instance, every node has zero distance with itself and thus every self pair must be joined. Our algorithms improve on this. In version one, for example, each index entry stores the minimum distance between two objects in the sub-tree. Thus it is quite likely that self pairs do not need to be joined.

Now we compare the two versions of our SRCP-tree. In our previous version, we can get a better performance than incremental join algorithm because it is likely that we can avoid self pair comparison. For example, assume the node pointed by entry S is the parent node of nodes pointed by entries A, B and C. When joining S with itself, in incremental join algorithm, six pairs need to be considered. They are AA, BB, CC, AC, AB, and BC. Version one of our structure is good in that we might eliminate AA, BB and CC according to the local closest pair information they stored. However, we can not eliminate AB, AC and BC, assuming $MinExistDist$ of each of the three pairs is larger than threshold T. The second version of our SRCP-tree can improve in this case. For instance, if $A.dist < B.dist$, and both objects in the closest pair stored at A are inside query range R, we can determine that there is no need to examine any pair involving B. This means we instantly avoid three pairs BB, AB, BC. Thus we believe the second version of the SRCP-tree has better query performance.

The other thing is, even though the two versions of the structure have exactly the same index size, the update cost for the second version is more expensive than the first version, since the second version needs to the sibling entries of every entry along the update path.

4.4 Extensions to Finding k Closest Pairs between Different Types of Objects

In order to find k closest pairs, we can use the same structure, but we modify the query algorithm as follows. We maintain an array of up to k different object pairs instead of one. The threshold we use to optimize the priority queue is the distance of the largest pair in the array.

In a practical scenario, we can have a data set which contains two different types of objects, e.g. high schools and primary schools, and we have an R-tree built on top of the data set. In this case, the query to find the closest pairs of high schools with primary schools is also supported by our techniques. We still augment the R-tree with some closest pair information, with the extension that each closest pair should contain one primary school and one high school, not the same type of schools.

5 Experimental Result

5.1 Experimental Setup

Since we know for sure that the range query followed by plane-sweep is not good especially when the query range is large, we implemented the heap algorithm [6], and the two versions of our SRCP trees. The query algorithm for each structure is implemented as we discussed in section 3 and section 4.

We implemented all of these structures in Java using XXL Library [20]. Our experiments were run on a PC with a 2.66-GHz Pentium 4 processor. Each tree node capacity is set to be between $m = 10$ and $M = 20$.

We test our experiments on two different data sets.

(1) We randomly generated data sets of size 20k, 40k, 60k, 80k with uniform-like distribution.

(2) We also use a real data set from US National Mapping Information web site (URL: http://mappings. usgs.gov/www/gnis/). We use the longitude and latitude of 26700 sites from Massachusetts as a 2D point data set. Notice that in the original file, some sites have the same position. We consider them as the same point. Here the 26700 is the number of total points.

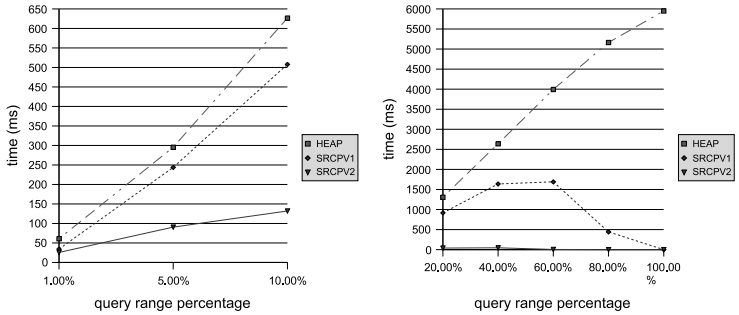
For each data set, we test each structure with query range 1%, 5%, 10%, 20%, 40%, 60%, 80% and 100% of the total area that the data set covers. We randomly generated 20 queries for each query range and the average running time is calculated.

(3) Furthermore, we experimented with objects with extents. For simplicity, we assume square objects. We use the 40K data set described in case (1) as the center of each object, while each object is extended as a square of fixed size. The object size is described as *area ratio*, which is defined as $\frac{O}{T} * 100\%$, where O is the area of an object and T is the area covered by the whole data set. Given a fixed query range 40% of T, we test each structure with area ratio 0.001%, 0.01%, 0.1%, 1% and 10%. For each ratio, 20 queries are randomly generated and the average running time is given.

5.2 Performance Comparison

We first compare the algorithms when the query range is small, between 1% and 10% of the space. As shown in Figure 3(a), the query time increases for all three structures. But both versions of SRCP outperform the heap algorithm. As the query range increases, more nodes will intersect the query range. Hence the comparison numbers and query times will increase. But due to the pre-computed information stored in both versions of SRCP trees, some branch can be eliminated before any calculation. That's why SRCP tree gains a better query performance in this situation. We also observe that the second version of the SRCP-tree has better performance.

When the query range is large, as shown in Figure 3(b), the query time for heap algorithm increases as it does for a small query range due to the same reason. However, the query time of SRCP trees decrease dramatically. This is



(a) Small query range comparison (b) Large query range comparison

Fig. 3. Different query range comparion for a 40K data set

because when the query range gets larger, the probability that a CP pair is in the query range also increases. The special case is when the query range is the same as the root node MBR. In this situation, we know for sure that the closest pair stored along with the root entry will be the desired pair. Hence the query time for it is almost zero.

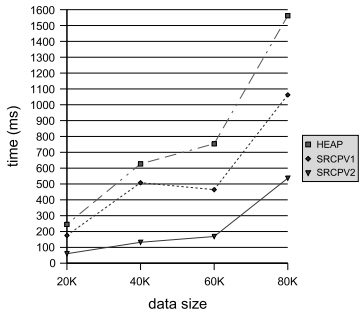


Fig. 4. Varying the number of objects.

We also compare the same query area ratio (10% of space) with different data sets. As we can see in Figure 4, when the size of a data set increases, the query time for the same area ratio will also increase, which is reasonable. When the data set contains more objects, the number of objects in the query range will be bigger. So there are more calculations. This causes the increase of query time when the data set size gets larger. However, we still observe that our approaches are much better than the existing approach, and that the second version of our structure is more robust.

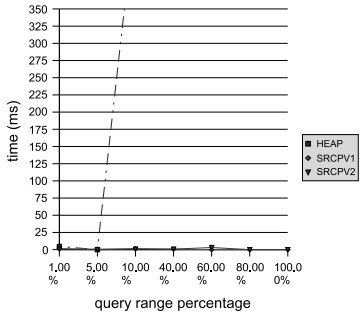


Fig. 5. Range query comparison for a real data set.

Figure 5 shows the query result in the real data set (the Massachusetts sites). In this experiment, SRCP trees outperform the heap algorithm even more than it does in the randomly generated data case. This is because in the real data set, many sites are very close to each other, especially some pairs has a distance almost zero. So in each page, one can always find a pair with very small distance. It is more like for SRCP query algorithm to stop when such a pair is found.

Our structure works for not only point data but also non-point data. Figure 6 is the query result of objects with extents. In this experiment, SRCP trees again have a better performance than heap algorithm. Notice that when the object area is enlarged, the performance of SRCP trees keeps almost unchanged while the query time of heap algorithm increases dramatically. In other words, SRCP trees outperform heap algorithm more when the object area gets larger.This is due to the fact that when object becomes larger, the MBR of the leaf node that contains this object is also augmented. The enlargement of the leaf node MBR will then cause the MBR of its parent node to be increased. The augmentation will be propogated till the root node. This senario makes the overlap between sibling MBRs happen more often. As we have seen, overlap is the key reason that gives the heap algorithm a poor performance while SRCP trees doesn't sacrifice too much from overlap.

Generally speaking, SRCP tree gains a much better performance on CP range query problem and index maintenance (the update algorithm is around three times slower) than heap algorithm with limited additional cost in space usage . That's a tradeoff between query performance and index maintainable time. Here we assume the maintenance is less often than range query, which is a practical assumption. Due to space limitations, we omit the performance for index update.

6 Conclusions and Future work

In this paper, we have extended the spatial closest-pair problem to involve a query range. This problem, although practical, has not been addressed in the literature. In particular, we consider the case where the closest pairs are supposed to come from a single index. We have proposed two versions of a new

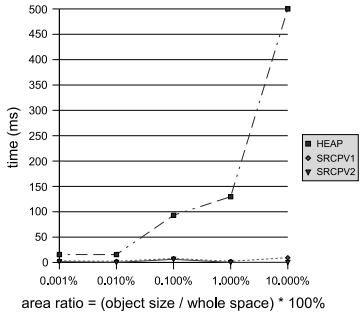


Fig. 6. Non-point data sets with different area ratio. A fixed value is used for the space covered by the data set.

index structure (the SRCP-tree) to solve this new problem. We compare our approaches with straightforward approaches, both analytically and experimentally. Performance results show that our methods are more robust than existing approaches. Among the two versions of our structures, the second version has overall the best query performance.

As for future directions, we are working on supporting the closest pair queries for multiple types of objects. For instance, the data set may maintain schools, bars, supermarkets, etc. The query may ask for k closest pairs of any two types of objects. One way to solve this problem by extending our solutions is that each index entry maintains, for each pair of different types, some local closest pair information. However, this is costly if there are many different types. We are seeking for more efficient solutions.

References

1. Oracle8 spatial cartridge. *http://technet.oracle.com/products/oracle8/info/sdods/xsdo7ds.htm.*
2. IBM informix spatial datablade module. *http://www-3.ibm.com/software/data/informix/ pubs/specsheets/SWSEC27152000D.pdf.*
3. N. Beckmann, H.-P. Kriegel, R. Schneider, and B. Seeger. The R*-Tree: an efficient and robust access method for points and rectangles. In *Proceedings of ACM/SIGMOD Annual Conference on Management of Data*, 1990.
4. S. Berchtold, B. Ertl, D. A. Keim, H.-P. Kriegel, and T. Seidl. Fast nearest neighbor search in high-dimensional space. In *Proceedings of the 14th International Conference on Data Engineering (ICDE)*, 1998.
5. A. Corral, Y. Manolopoulos, Y. Theodoridis, and M. Vassilakopoulos. Closest pair queries in spatial databases. *Technical Report, Aristotle Univ. of Thessaloniki, Greece, url=http://delab.csd.auth.gr/∼michaliz/cpq.html*, 1999.
6. A. Corral, Y. Manolopoulos, Y. Theodoridis, and M. Vassilakopoulos. Closest pair queries in spatial databases. In *Proceedings of ACM/SIGMOD Annual Conference on Management of Data*, 2000.
7. M. Dietzfelbinger, T. Hagerup, J. Katajainen, and M. Penttonen. A reliable randomized algorithm for the closest-pair problem. *Journal of Algorithms*, 25(1), 1997.

8. V. Gaede and O. Günther. Multidimensional access methods. *ACM Computing Surveys*, 30(2), 1998.
9. A. Guttman. R-trees: a dynamic index structure for spatial searching. In *Proceedings of ACM/SIGMOD Annual Conference on Management of Data*, 1984.
10. G. R. Hjaltason and H. Samet. Incremental distance join algorithms for spatial databases. In *Proceedings of ACM/SIGMOD Annual Conference on Management of Data*, Seattle, WA, USA, 1998.
11. G. R. Hjaltason and H. Samet. Distance browsing in spatial databases. *ACM Transactions on Database Systems (TODS)*, June 1999.
12. N. Katayama and S. Satoh. The SR-tree: An index structure for high-dimensional nearest neighbor queries. In *Proceedings of ACM/SIGMOD Annual Conference on Management of Data*, 1997.
13. G. Kollios, D. Gunopulos, and V. J. Tsotras. Nearest neighbor queries in a mobile environment. In *Proceedings of the International Workshop on Spatio-Temporal Database Management*, pages 119–134, 1999.
14. F. Korn and S. Muthukrishnan. Influence sets based on reverse nearest neighbor queries. In *Proceedings of ACM/SIGMOD Annual Conference on Management of Data*, 2000.
15. N. Roussopoulos, S. Kelly, and F. Vincent. Nearest neighbor queries. In *Proceedings of ACM/SIGMOD Annual Conference on Management of Data*, 1995.
16. T. Seidl and H.-P. Kriegel. Optimal multi-step k-nearest neighbor search. In *Proceedings of ACM/SIGMOD Annual Conference on Management of Data*, 1998.
17. M. Smid. Closest point problems in computational geometry. In J.-R. Sack and J. Urrutia, editors, *Handbook on Computational Geometry*. Elsevier Science Publishing, 1997.
18. Z. Song and N. Roussopoulos. K-nearest neighbor search for moving query point. In *Proceedings of the 7th International Symposium on Spatial and Temporal Databases*, pages 79–96, 2001.
19. I. Stanoi, D. Agrawal, and A. E. Abbadi. Reverse nearest neighbor queries for dynamic databases. In *Proceedings of the ACM SIGMOD Workshop on Research Issu in Data Mining and Knowledge Discovery*, 2000.
20. J. van den Bercken, J.-P. Dittrich B. Blohsfeld, J. Krämer, T. Schäfer, M. Schneider, and B. Seeger. XXL- a library arrproach to supporting efficient implementations of advanced database queries. In *Proceedings of the 27th VLDB Conference*, 2001.
21. C. Yang and K. Lin. An index structure for improving closest pairs and related join queries in spatial databses. In *Proceedings of the International Database Engineering and Applications Symposium (IDEAS'02)*, 2002.
22. C. Yang and K.-I. Lin. An index structure for efficient reverse nearest neighbor queries. In *Proceedings of the 14th International Conference on Data Engineering (ICDE)*, 01.
23. D. Zhang, V. J. Tsotras, and B. Seeger. A comparison of indexed temporal joins. *Tech Report, UCR-CS-00-03, CS Dept., UC Riverside, url=http://www.cs.ucr.edu/ ∼donghui/publications/tempjoin.ps*, 2000.
24. D. Zhang, V. J. Tsotras, and B. Seeger. Efficient temporal join processing using indices. In *Proceedings of the 14th International Conference on Data Engineering (ICDE)*, 2002.