Finding k-Closest-Pairs Efficiently for High Dimensional Data *

Mario A. Lopez *Swanwa Liao*
Department of Mathematics and Computer Science
University of Denver, Denver CO 80208.
{mlopez,swliao}@cs.du.edu

Abstract

We present a novel approach to report approximate as well as exact k-closest pairs for sets of high dimensional points, under the L_t-metric, $t = 1, \ldots, \infty$. The proposed algorithms are efficient and simple to implement. They all use multiple shifted copies of the data points sorted according to their position along a space filling curve, such as the Peano curve, in a way that allows us to make performance guarantees and without assuming that the dimensionality d is constant. The first algorithm computes an $O(d^{1+1/t})$ approximation to the k^{th} closest pair distance in $O(d^2 n \log + dk(d + \log k))$ time. Experimental results, obtained using various real data sets of varying dimensions, indicate that the approximation factor is much better in practice. In the second algorithm we use this approximation in order to find the exact k closest pairs in $O(dM)$ additional time, where M is the number of points in certain short subsegments of the space-filling curve. The exact algorithm is particularly efficient and $M = O(k)$ can be guaranteed, when presented with data sets that satisfy certain separation conditions. The proposed approach can be easily adapted to other proximity problems, including fixed-radius neighbor search, minimal k-point clustering, and nearest neighbor search.

1 Introduction

The k-closest-pairs (CP_k) problem can be stated as follows: given a set P of n distinct points in $\mathbb{R}^d$ and positive integers k and t compute the k closest pairs of P under the L_t metric. The L_t distance $d_t(p, q)$ of $p, q \in \mathbb{R}^d$ is defined as $(\sum_{i=1}^{d} |p_i - q_i|^t)^{1/t}$ for $1 \leq t < \infty$ and $\max_{1 \leq i \leq d}\{|p_i - q_i|\}$ for $t = \infty$, where p_i denotes the i^{th} coordinate of p.

The CP_k problem when $d > 2$ is an arbitrary but fixed constant has been studied extensively in the past. Shamos and Hoey [24] and Bentley and Shamos [3] solved CP_1 in $O(n \log n)$ time. Schwarz *et al* [23] studied the on-line version of the same problem also for $d > 2$. Salowe [21] shows how to compute the k^{th} closest pair in $O(n \log^d n)$ time under L_∞. The first result for enumerating more than one pair was by Smid [25] who solved $CP_{n^{2/3}}$ in $O(n \log n)$ time. Lenhof and Smid [19] followed with a solution to the general CP_k problem that runs in $O(n \log n + k)$ time. Unfortunately, when d is not viewed as a constant, all of the algorithms reported above suffer from the so called "dimensionality curse" – the constant factors hidden in the O-notation grow exponentially with d. The $O(n \log^d n)$ result clearly suffers from the curse. For others, closer inspection reveals the dependency of the complexity on a factor of the form c^d, for some constant $c > 1$, hidden in the notation. Clearly, a simple brute force algorithm can solve the problem in $O(dn^2)$ time. Although we may often assume that d is much smaller than n, in practice however, c^d can quickly overcome a reasonably large n, even for very small d, thus making the brute force solution the preferred choice. For example the algorithm in [19] consists of two phases: an approximation phase in which $20^d(k + 2n) \leq \binom{n}{2}$ must be satisfied, followed by an enumeration phase in which the processing time is at least $\Omega(5^d k + 4^d n \log n)$ in the worst case. For the best scenario of $k = 1$ in the approximation phase, d cannot exceed $\log_{20} n$. This bound gets progressively smaller as k increases. Even for fairly large n, say 10 billion, d cannot be more than seven for any k. When this condition is violated, the algorithm degenerates to a brute force $O(n^2)$ algorithm. Furthermore, as d increases and for practical values of n, the time for the enumeration phase quickly becomes $\Omega(n^2)$ even for moderately small d.

The results so far suggest that an exact solution to the CP_k problem that is polynomial in d may be difficult to find. The problem is aggravated for many data distributions (including uniform) where, as d increases, there may be no significant difference between distance of the closest pair and the k^{th} closest pair, possibly rendering the solution meaningless. This is a consequence of the results of [6]. Thus, for "meaningful" data, it is only natural that an exact algorithm exploits, for its efficiency, the natural separation conditions of the data. In the methods mentioned above

*This work has been partially supported by the National Science Foundation under grant IRI-9610240.

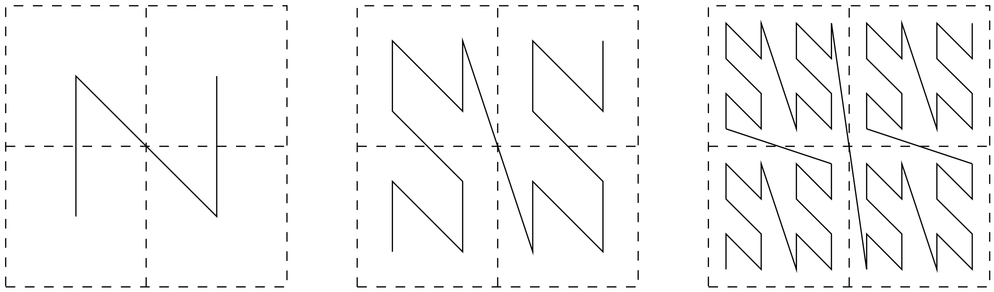

Figure 1: The first three stages of Peano curve approximation.

the exponential growth in d continues regardless of the data distribution. On the other hand, the exact CP_k algorithm proposed in this paper comes with good performance guarantees for data sets that satisfy certain fairly general separation conditions. The proposed algorithm utilizes multiple shifted copies of P ordered along a space-filling curve and consists of two phases: an *Approximation Phase* where the k^{th} closest pair distance is approximated and guaranteed to be within a small factor from the true solution, followed by an *Exact Phase* in which the actual k-closest pairs are found. As our experimental results indicate, the approximate solution is very good and may suffice, particularly for applications where the best choice of metric is not obvious (as is the case when trying to determine the "distance" between two binary images, for example).

In our analyses, in order to make the dependency on d explicit, the space requirement for P is $O(dn)$ and the processing time to visit every coordinate of a point is $O(d)$ (instead of the more familiar $O(n)$ and $O(1)$ respectively). Thus, a simple brute force method would take $O(dn^2)$ time and $O(dn)$ space. Furthermore, to simplify proofs, we assume that the input data has been scaled to $[0..1)^d$. Generalization to other extents is trivial. In particular, the algorithms presented in this paper can easily be modified to support integer-only arithmetic so as to achieve greater computational efficiency in practice.

The rest of the paper is organized as follows. In Section 2 we review Peano curves and discuss some of their properties which are needed to analyze the correctness and efficiency of our algorithms. In Sections 3 and 4 we derive and analyze the approximation and exact algorithms, respectively. Experimental results using real data sets of different size and dimensionality are presented in Section 5. Generalizations to other proximity problems are discussed in Section 6. Section 7 concludes.

2 Preliminaries

Algorithms proposed in this paper are based on the use of space-filling curves that span a finite hypercube I^d in $\mathbb{R}^d$. For convenience and ease of implementation, we describe our results in terms of Peano (also called Z-order) curves. Other choices, such as Hilbert curves, are also possible.

The power of these curves lies in their ability to linearly order multidimensional points into nested compartments such that points within a compartment are also close in the linear order. In this paper we take advantage of this property.

The use of Peano curves is not new; they have been used in a variety of applications including quadtree representation and R-tree packing [15, 22].

The construction of the Peano curve can be viewed as a sequence of stages. The output Z_i^d of stage i is a curve that approximates the ideal Peano curve. At each stage, an instance of the curve of the previous stage is scaled and placed in each of the equal-sized 2^d sub-quadrants. These sub-quadrants are then visited in lexicographic order, connecting the endpoints of the 2^d sub-curves, in order to produce the continuous curve at the next stage. The process continues indefinitely to produce the ideal Peano curve. Figure 1 illustrates the first three stages of curve approximation, for the square I^2. Of course, the ideal Peano curve cannot be constructed on a finite precision machine. This is not a problem as we only care about the *relative position* of points along the curve, as given by a relation $<_z$. Thus, the ideal curve or the positions of points on this curve need not be computed explicitly. See [15] for details on how to implement the $<_z$ relation when given two d-dimensional points. This simple to code function, as presented in [15], is based on a finite sequence of "quadrant containment" comparisons. It can be implemented to run in $O(d)$ time on a finite precision machine, i.e., one where the number of bits ρ needed to represent the underlying coordinates in standard binary form is bounded by a constant. This assumption is reasonable in practice given the finite representation of numbers used by most language implementations. For example, under IEEE Standard 754, $\rho = 32$ for integer, and $\rho = 126$ for single-precision floating point. For sake of simplicity, we provide our analyses assuming that ρ is constant. Thus, we assume that a single evaluation of $<_z$ requires $O(d)$ time. This is similar to the widespread assumption used when analyzing algorithms that a simple scalar arithmetic operation requires $O(1)$ time. It is not difficult to modify the resulting complexity claims using explicit functions of ρ.

Remember that the input data is assumed to fit in $[0,1)^d$.

Definition 1 *an r-region is an open ended hypercube in $\mathbb{R}^d$ with side length of $r = 2^{1-m}$ having the form: $\Pi_{i=0}^{d-1}[a_i r, (a_i+1)r)$, where each a_i and m are in $\mathbb{N}$.*

As one can see from Figure 1, for $d = 2$ and $I = [0..1)$, there is a clear pattern emerging from the approximating Peano curves at various stages. At the i^{th} stage, for any $j = 0, \dots, i$, one can view the initial square as divided into a sequence of 2^{2j} equal-sized smaller squares. Each of these squares is an r-region with $r = 2^{-j}$. Vertices in the same r-region are linked into a contiguous chain, with both ends linked to vertices from other r-regions (except for the first and last r-regions, of course). In the example of Figure 1, we have used dashed lines to illustrate the four $\frac{1}{2}$-regions of $[0,1)^2$. For higher dimensions, the recursive subdivision process is similar. At stage $i \geq 0$ the initial square $[0,1)^d$ consists of 2^{dj} disjoint r-regions with side length $r = 2^{-j}$, $j = 0, \dots, i$.

The key to the efficiency of our algorithms is the use of multiple shifted copies of P, sorted in Peano order, in a way that allows us to make performance guarantees without assuming d is constant. We formally define these sorted lists as follows.

Definition 2 *Let $\{Z_j, j = 0, ..., d\}$, be the $(d+1)$ Peano ordered lists of points of P such that $Z_j = \{P[i] + v^{(j)}\}$, $i = 1, \dots, n$, and $Z_j[i] <_z Z_j[i+1]$, where $v^{(j)} = (j/(d+1), \dots, j/(d+1)) \in \mathbb{R}^d$. We call this set of lists the ZL-set.*

Note that the points of each Z_j are now in $[0..2)^d$ and we are interested in r-regions with $r = 2^{1-h}$, $h = 0, 1, 2, \dots$.

Clearly the ZL-set can be constructed in $O(d^2 n \log n)$ time. Furthermore, the space required by the ZL-set can be bounded by $O(dn)$ by using pointers to a master array P, where the actual coordinates of the input points are kept. Shifts are then applied to individual points in real-time as needed.

We now discuss some properties of a Peano-sorted list Z.

Observation 1 *If we divide $\mathbb{R}^d$ into equal sized r-regions, for any r, then each r-region contains one and only one contiguous segment of a Peano-sorted list Z.*

The following two observations are simple consequences of the above:

Observation 2 *let $R_i(j)$ denote the size of the smallest r-region that contains two points $Z[i]$ and $Z[i+j]$ of Z, $1 \leq i < i+j \leq n$. Then for each fixed i, $R_i(j)$ is a monotone increasing function of j.*

Observation 3 *For given r, the number of points of Z inside the r-region that contains q is simply $e - b + 1$ where b and e are the indices of the points bounding the r-region in Z. These indices can be found in $O(d \log n)$ time.*

Indices b and e can be found by a simple binary search. We illustrate how to find b. The value e can be found similarly. let s be the pivotal position at the i^{th} iteration of a binary search. If $q <_z Z[s]$, then $b < s$. Else, if $Z[s]$ is in the r-region, then $b \leq s$, else $b > s$.

The following lemma from Chan [8] is fundamental in proving the efficiency of our algorithms. In this lemma, a point p is said to be α-central in an r-region iff for each $i = 0, \dots, d-1$, we have $\alpha r \leq \mathrm{mod}(p_i, r) < (1-\alpha)r$, where $\mathrm{mod}(p_i, r) = p_i - \lfloor p_i/r \rfloor r$.

Lemma 1 *Suppose d is even. For any point $p \in \mathbb{R}^d$ and $r = 2^{-m}$ ($m \in \mathbb{N}$), there exists $j \in \{0, \dots, d\}$ such that $p + v^{(j)}$ is $(1/(2d+2))$-central in its r-region cell.*

For the rest of the paper $C_\varepsilon(q)$ denotes the closed hypercube C about a point q with orthogonal radius ε. We can now prove a result that will be useful in finding small r-regions that contain all pairs involving a given input point q.

Theorem 2 *Suppose d is even. Given a point $q \in \mathbb{R}^d$ and ε, where $0 < \varepsilon < 1/(2d+2)$, there exists j, $0 \leq j \leq d$, such that $C_\varepsilon(q + v^{(j)})$ is contained in an r-region with r satisfying $r/(4d+4) \leq \varepsilon < r/(2d+2)$.*

Proof. Note that $r/(4d+4) \leq \varepsilon < r/(2d+2)$ defines a unique r. If we divide $\mathbb{R}^d$ into equal sized r-regions with r as defined above, then Lemma 1 guarantees the existence of j such that $q' = q + v^{(j)}$ is $(1/(2d+2))$-central in its r-region. Hence, the hypercube $C_{r/(2d+2)}(q')$ is completely contained in this r-region. But by construction, $\varepsilon < r/(2d+2)$, so $C_\varepsilon(q')$ is also contained in the r-region. ∎

Theorem 2 basically states that if a closed hypercube C is shifted $d+1$ times as outlined above, and if we were to find a tightest r_j-region that encloses the shifted C for each shift, then $\min\{r_0, \dots, r_d\}$ is bounded above by a small value. Note that for given q and r, the r-region of q can be easily found in $O(d)$ time. Henceforth, we assume d is even (for odd d, simply replace d by $d+1$).

function *Approx*(P, k)
  Build the ZL-set of P
  Initialize Q with k arbitrarily chosen pairs
  foreach list $Z = \langle z_1, \ldots, z_n \rangle$ of the ZL-set **do**
    for $i \leftarrow 1$ **to** $n-1$ **do**
      for $j \leftarrow (i+1)$ **to** $\min(i+k, n)$ **do**
        Find the smallest τ-region that contains (z_i, z_j) in Z
        Let i' and j' denote the positions of the unshifted z_i and z_j in P, resp.
        if $\tau <$ *Max*(Q) **then**
          if *In*(i',j',Q) **then**
            Update(i', j', τ, Q)
          else
            DeleteMax(Q)
            Insert(i', j', τ, Q)
        else
          Stop pairing for z_i and proceed with next i
  return $\max_{(p,q)\in Q} \{\text{dist}_t(p,q)\}$
end

Figure 2: The approximation algorithm.

3 The Approximation Phase

In this section, we develop an approximation algorithm for the CP_k problem. For any t, the algorithm finds a value $\delta_{k,t}$ that approximates the true k^{th} pair distance $\delta^*_{k,t}$ from above and that is within a small factor of $\delta^*_{k,t}$. This approximation is computed from the k smallest r-regions enclosing unique pairs of points from P. We show how to find these r-regions without considering all pairs of points. For some applications the approximate k-pairs found are good enough. Otherwise, in the next section we describe how to use $\delta_{k,t}$ to find the exact solution.

The approximation algorithm uses an "enhanced" priority queue Q of size k built on the size of the smallest r-region that encloses selected shifted pairs from P. More specifically, the following operations are required:

1. *Max*(Q): returns the largest value (r-region) in Q.
2. *DeleteMax*(Q): remove from Q the entry with the largest value (r-region).
3. *Insert*(i,j,τ,Q): insert the pair $(P[i], P[j])$ with associated value τ.
4. *In*(i, j, Q): return *true* iff the pair $(P[i], P[j]) \in Q$.
5. *Update*(i,j,τ): update the value associated with pair $(P[i], P[j])$ to τ provided that the pair's current value is higher than τ.

Each entry of Q stores a pair of unshifted points from P as well as the size of the smallest r-region that contains the (shifted) pair. Q always contains k entries and is implemented as a binary search tree ordered lexicographically on the indices of the unshifted pairs. The tree is augmented so that each node v stores a pointer to the smallest r-value stored in a node of the subtree whose root is v. The operations can then be implemented in $O(\log k)$ time except for *Max* which runs in $O(1)$ time.

The approximation algorithm derives its efficiency from Observation 2. It is based on finding efficiently the k smallest enclosing r-regions for distinct pairs of points from P. Observation 2 implies that this can be done by examining each point of the ZL-set and up to k of its successors. At all times, Q stores the best k distinct pairs found so far. Since each point appears once in each list of the ZL-set, a pair may be considered up to $d+1$ times and we need the capability of updating its cost. Upon completion, the approximation δ_k is simply the largest L_t distance for the k pairs in Q. See Figure 2 for a more detailed description.

We now present the main results of the paper.

Theorem 3 $\delta^*_{k,t} \leq \delta_{k,t} \leq d^{1/t}(4d+4)\delta^*_{k,t}$.

Proof. Since δ_k was calculated from k distinct point pairs, clearly $\delta^*_k \leq \delta_k$. For the rest of the inequality, since any pair distance must be less than $d^{1/t}$, if $\delta^*_k \geq 1/(4d+4)$ then $\delta_k \leq d^{1/t}(4d+4)\delta^*_k$ is trivially satisfied. Thus, we only need to consider the case when $\delta^*_k < 1/(4d+4)$.

Let $S = \{(s_1, q_1), \ldots, (s_k, q_k)\}$ be the actual solution pairs to the given CP_k instance and let $x_i = dist_t(s_i, q_i)$, $1 \leq i \leq k$. Note that $\delta_k^* = \max\{x_1, \ldots, x_k\}$. Since $x_i \leq \delta_k^* < 1/(4d+4)$, for each $1 \leq i \leq k$ Theorem 2 guarantees the existence of $0 \leq j_i \leq d$ such that $C_{x_i}(s_i + v^{(j_i)})$ is contained in an r_i-region, where $r_i \leq x_i(4d+4)$. It follows that $q_i + v^{(j_i)}$ belongs to that r_i-region. Let $r_{\max} = \max\{r_1, \ldots, r_k\}$. If $Max(Q) \leq r_{\max}$ then each pair (p, q) in Q satisfies $\mathrm{dist}_t(p, q) \leq d^{1/t} r_{\max} \leq d^{1/t}(4d+4)\delta_k^*$. In other words, $\delta_k \leq d^{1/t}(4d+4)\delta_k^*$, as needed. Accordingly, we now show that $Max(Q) \leq r_{\max}$. For ease of notation, let $s_i' = s_i + v^{(j_i)}$ and $q_i' = q_i + v^{(j_i)}$ and, without loss of generality, assume that q_i' is positioned after s_i' in the sorted Z_{j_i}.

Suppose that every pair (s_i', q_i'), $1 \leq i \leq k$, is considered for insertion into Q. Then clearly $Max(Q) \leq r_{\max}$. Otherwise, there exists an i such that (s_i', q_i') was not examined by the algorithm when processing list Z_{j_i}. This can happen in two cases. First, the algorithm examined s_i' paired with all its k neighbors but q_i' was more than k positions away from s_i'. By Observation 1, these k neighbors paired with s_i' must also be contained in the same r_i-region as q_i'. Hence the algorithm found at least k pairs whose r-value is no larger than r_i, and thus no larger than $r_{\max}$. Second, the algorithm encountered a successor g of s_i' that occurs before q_i' and such that the r'-region enclosing (s_i', g) is no smaller than $Max(Q)$. Again by Observation 1, we have $r' \leq r_i$. Thus, we have $Max(Q) \leq r' \leq r_i \leq r_{\max}$ which concludes the proof. ■

Now we analyze the running time of the algorithm. To simplify the analysis we assume each point is paired with k successors. (In practice, and depending on the position of points on the sorted lists and the value of k, much fewer successors are actually considered.) First the ZL-set is built in $O(d^2 n \log n)$ time. For each pairing, it takes $O(d + \log k)$ time to compute the smallest enclosing r-region and, if necessary, to update Q. Thus, the running time is $O(d^2 n \log n + dnk(d + \log k))$. Below we show that this bound is actually a lot smaller when using fixed precision representation of numbers.

In the proof of Theorem 4 below, we say that queue Q is *improved* if one of its r-values is made smaller. Let 2^ρ (resp. $2^{-\rho}$) be the largest (resp. smallest) positive integer (resp. floating point) value representable on a fixed precision machine. Since each r-region is such that $r = 2^m$ (resp. $r = 2^{1-m}$), $m \in \mathbb{N}$, each r-value of Q can take on at most $\rho + 1$ (resp. $\rho + 2$) different values. This implies that Q, having k nodes, can be improved at most $O(k\rho)$ times, where ρ is a constant – an assumption discussed in Section 2.

Theorem 4 *The approximation algorithm runs in $O(d^2 n \log n + dk(d + \log k))$ time and uses $O(dn + k)$ space.*

Proof. We partition the $O(dnk)$ pairs examined by the algorithm into three disjoint sets S_1, S_2, and S_3 as follows. Let (s, q) be a pair under consideration and let r_{sq} be the size of the smallest r-region containing s and q. If $r_{sq} \geq \mathrm{Max}(Q)$, then (s, q) is added to S_1. Otherwise, if (s, q) exists in Q but does not improve Q, then it is added to S_2; else it is added to S_3. Line 15 of the algorithm implies that $|S_1| = O(dn)$. Furthermore, since all pairs coming from the same list of the ZL-set are unique, each such list can contribute at most k pairs to S_2. Thus, $|S_2| = O(dk)$. Finally, we note that each pair of S_3 improves Q. Since Q cannot be improved more than $O(\rho k)$ times, if ρ is considered a constant independent of d and n, then $|S_3| = O(k\rho) = O(k)$.

Each pair in S_1, S_2, and S_3 requires $O(d)$ time for the r-region calculation, with pairs from S_2 and S_3 requiring an additional $O(\log k)$ time for manipulating Q. Thus, the total processing time, including the $O(d^2 n \log n)$ time spent for building the ZL-set, is $O(d^2 n \log n + d^2 n + (dk + k)(d + \log k))$ or $O(d^2 n \log n + dk(d + \log k))$ as claimed.

The algorithm requires $O(dn)$ space for P and the ZL-set and $O(k)$ space for the queue. Thus the total space requirement is $O(dn + k)$. This concludes the proof. ■

Note that if $k = O(\min(n \log n, dn \log n / \log k))$ then the algorithm runs in $O(d^2 n \log n)$ – the same time bound required for constructing the ZL-set. This includes $k = O(n)$.

4 The Exact Phase

The Approximation Phase produced an approximation $\delta_{k,t}$ to the exact k^{th} closest pair distance $\delta_{k,t}^*$ such that $\delta_{k,t} \leq d^{1/t}(4d+4)\delta_{k,t}^*$. In order to find the actual k closest pairs we identify for each point $q \in P$ a small r-region R such that the shifted $C_{\delta_{k,t}}(q)$ is inside R and enumerate all points inside R paired with q. Once this process has been applied to all points of P we select the best k out of M enumerated pairs. The whole process takes $O(d^2 n \log n + dM)$ time. If desired, our algorithm can be easily modified to compute in $O(d^2 n \log n)$ time the value M. This allows the user to confirm, abort, or use a smaller k before proceeding with a (potentially expensive) enumeration task.

Consider a point $q \in P$. We wish to find all points in P whose distances from q do not exceed $\delta_{k,t}$. To accomplish this we identify an r-region that contains $C_{\delta_{k,t}}(q + v^{(j)})$ for some j. Furthermore, we want this r-region to contain the fewest number of points. To identify such region we examine all $d+1$ lists of the ZL-set. For each $0 \leq j \leq d$ we identify in $O(d)$ time the smallest r_j-region that encloses $C_{\delta_{k,t}}(q + v^{(j)})$. By Observation 3 and for each j, we can find in $O(d \log n)$ time the boundaries of the contiguous segment of Z_j

inside the given r_j-region. Let M_j be the number of points in that segment, which can be calculated in constant time once the boundary indices are known. Upon completion, let i be such that $M_i = \min\{M_0, ..., M_d\}$ and for each point $s + v^{(i)}$, $s \neq q$, of Z_i inside the corresponding r_i-region, if $\text{dist}_t(s, q) \leq \delta_{k,t}$, then the pair distance and their indices in P are saved for latter processing. Subsequently, q is removed from all lists in the ZL-set.

Once all points of P are processed by the above algorithm, we have identified M pairs (note that M is simply the sum of the ensuing M_i's) of P with each pair distance less than or equal to $\delta_{k,t}$. We now use a linear time selection algorithm to identify the k^{th} smallest and in the process, identify all k closest pairs. This concludes the Exact algorithm. We summarize the above discussion in the following lemma.

Lemma 5 *Given the ZL-set and the approximate k^{th} closest distance $\delta_{k,t}$, then the exact phase algorithm finds all k nearest pairs in $O(d^2 n \log n + dM)$ time using $O(M)$ additional space.*

Note that when storage is limited one can eliminate the space dependency on M by using a heap of size k at the expense of $O(d^2 n \log n + dM \log k)$ processing time.

Consider the graph of $\delta^*_{h,t}$ as a function of h. It is natural to ask which characteristics of the data (i.e., of this function) would guarantee that M is small (ideally, $M = O(k)$), particularly in light of the comments made in the Introduction. We would expect that steepness of the function in the neighborhood of k (i.e., a large value of $\delta^*_{k+1,t} - \delta^*_{k,t}$) would greatly contribute to lowering M. We show that a much less restrictive separation condition suffices in order to guarantee efficiency.

Theorem 6 *If there exists a $k' = O(k)$, such that $\delta^*_{k,t} \leq \delta^*_{k',t}/(d^{1/t}(4d+4))^2$, then the exact phase requires $O(d^2 n \log n + dk)$ time. In other words, $M = O(k)$ in Lemma 5.*

Proof. Theorem 3 guarantees $\delta_{k,t} \leq d^{1/t}(4d+4)\delta^*_{k,t}$. Together with the fact that $\delta^*_{k',t} < d^{1/t}$ it follows that $\delta_{k,t} < 1/(4d+4)$. Consider now an arbitrary point $q \in P$. Since $\delta_{k,t} < 1/(4d+4)$, Theorem 2 guarantees that there exists a j such that $C_{\delta_{k,t}}(q+v^{(j)})$ is contained in an r_j-region with $r_j/(4d+4) \leq \delta_{k,t}$. Together with Theorem 3 this implies that $r_j \leq d^{1/t}(4d+4)^2\delta^*_{k,t}$ which in turn, by the assumptions of this theorem, implies that $r_j d^{1/t} \leq \delta^*_{k',t}$. Thus, when we traverse a segment in Z_j that contains a superset of all points of P that are within distance $\delta^*_{k,t}$ of q, every point s thus visited satisfies $\text{dist}_t(q + v^{(j)}, s) < r_j d^{1/t} \leq \delta^*_{k',t}$. Therefore each such point is also a solution pair for $CP_{k'}$. (Note that not all solution pairs of $CP_{k'}$ are necessarily visited this way.) After applying the same argument to all points of P, it must be the case that $M \leq k'$ Thus, our algorithm runs in $O(dn \log n + dk') = O(dn \log n + dk)$ time, as claimed. ■

Theorem 6 says that, as a function of h, $\delta^*_{h,t}$ can be fairly flat around the target k and we can still guarantee good performance provided that "eventually", for large enough k', $\delta^*_{k',t} - \delta^*_{k,t}$ is large. In fact, our experiments on various data sets have shown that good results can be obtained quickly even if the separation condition of Theorem 6 is much further relaxed.

Finally, for our algorithm to make sense, we must have $d < n/\log_2 n$ (as otherwise the performance degenerates to that of brute force). This is less restrictive than $d < \log_{20} n$ as required by [19].

5 Experimental Results

We implemented the approximation algorithm of Section 3 using the gnu C++ compiler on a Sun UltraSparc 2. We tested the quality of the approximation returned by our algorithm for various k using three real data sets: COT ($d = 16$, $n = 68,040$), CH ($d = 32$, $n = 68,040$)[1], and Landsat ($d = 60$, $n = 275,465$)[2]. Each set corresponds to a collection of real images. Each image is represented as a d-dimensional vector whose coordinates represents standard features extracted from the image. Bhanu *et al* [7], for example, describe how to represent and measure distance between images using feature vectors.

Below we show that although in theory we can only guarantee the distance of the k^{th} pair found is within $O(d^{1+1/t})$ factor from the distance of the actual k^{th} solution pair, in practice however, the approximate results are much better (we obtain an approximation factor of about 2). Furthermore, this factor is even better when we consider the average ratio over *all* k approximate closest pairs found.

The experimental results using L_2 are summarized in Table 1. We use $d_{appx,i}$ and $d_{true,i}$ to denote the distances of the i^{th} approximate pair found and the exact i^{th} solution pair, respectively. Specifically, for each data set, the table shows the ratio $\frac{d_{appx,k}}{d_{true,k}}$ for various values of k. For each such k, the average ratio over all k approximate pairs found; i.e., $(\Sigma_{i=1}^{k} \frac{d_{appx,i}}{d_{true,i}})/k$ is also shown (in parentheses).

As shown, the approximate solutions found by our approximate algorithm are much better than what the theory predicts. Thus, for applications in which the metric is chosen heuristically our algorithm is a good alternative.

[1] Sets COT and CH were obtained from `http://kdd.ics.uci.edu/databases/CorelFeatures/ CorelFeatures.html`.

[2] Data set was obtained from `http://vivaldi.ece.ucsb.edu/users/wei/gaborfeatures/airphotoFeatures`.

Data Set	Size (n)	Dim	$k = 1$	$k = 10$	$k = 20$	$k = 50$	$k = 100$
COT	68,040	16	1.24	1.67 (1.60)	1.82 (1.65)	1.93 (1.67)	1.96 (1.68)
CH	68,040	32	1.00	1.00 (1.00)	1.32 (1.03)	1.15 (1.03)	1.68 (1.10)
Landsat	275,465	60	1.32	2.33 (1.57)	2.23 (1.54)	2.10 (1.43)	1.98 (1.35)

Table 1: The k^{th} and average of all k (in parentheses) approximation ratios for three real data sets.

6 Other Proximity Problems

Based on the results of Theorem 2 we are developing a unified approach for solving various proximity problems for high-dimensional data. The approach is similar to that used in the previous sections and uses the same ZL-set. We briefly describe preliminary results on a few sample problems. Complete details and further results will be published in a full version of this paper.

Consider first the Fixed Radius Near Neighbors ($FRNN_\varepsilon$) problem: given a set P of n points in $\mathbb{R}^d$ and a distance ε, find all pairs of points of P which are no further apart than ε under L_t. Previous work on this problem includes [4, 9, 12, 26]. All of these (if applicable to high d) suffer from the dimensionality curse. We can solve the $FRNN_\varepsilon$ problem by simply removing portions of the CP_k algorithm presented earlier. First, eliminate the approximation phase of the CP_k algorithm, and instead use ε in place of $\delta_{k,t}$. Second, out of the $O(M)$ pairs examined simply report those whose distance does not exceed ε. It is easy to show that the $FRNN_\varepsilon$ algorithm runs in $O(d^2 n \log n + dM)$ time. As in the CP_k algorithm, M can be found in advance in $O(d^2 n \log n)$ time. This allows the user to input a smaller ε if M is too large, or to abort the query. Finally, $M = O(k)$ can be guaranteed when the input data sets satisfies separation conditions similar to those discussed in Section 4.

Consider now the Smallest k-point Clustering (SC_k) problem: given a set P of n points in $\mathbb{R}^d$ and $1 < k < n$, find a subset of P of size k that is contained in a d-dimensional L_t-sphere of smallest radius. The SC_k problem for $d > 2$ has been studied by various researchers [3, 11, 14]. In [11], the most recent of these, the authors partition the original problem into $O(n/k)$ subproblems, and each subproblem is solved in $O(k^{d/2} \log^2 k)$ time (the authors treat d as a constant). This time is prohibitive even for moderate d. Previous results suggest that the SC_k problem is probably very difficult, even for reasonably small values of d. Thus, a good and efficient approximation algorithm could be very beneficial. Consider the following approach. For each list Z_i of the ZL-set, $i = 0, \ldots, d$, find the smallest r-region containing $Z_i[j]$ and $Z_i[j+k-1]$, $j = 1, \ldots, n-k+1$. Upon completion, let $L_a[b]$ and $L_a[b+k-1]$ be the pair whose enclosing r-region is smallest among all pairs considered by the algorithm in all $d+1$ lists. Let $r\prime$ be the the size of this smallest r-region. Our approximate solution is simply the set $\{L_a[b], \ldots, L_a[b+k-1]\}$. This algorithm runs in $O(d^2 n \log(n))$ time. Furthermore, let δ_t^* be the radius of the smallest L_t-sphere containing the actual solution set and $\delta_t = d^{1/t} r'/2$, then $\delta_t^* \le \delta_t \le d^{1/t}(2d+2)\delta_t^*$. Complete details will be available in the full version of the paper.

Lastly, we briefly discuss the Nearest Neighbor (*NN*) problem: given a set P of n points in $\mathbb{R}^d$, find the point of P closest to an arbitrary query point q under L_t. Efficient *NN* searching for high d is an important problem in many application areas such as pattern classification, information retrieval, multimedia and voice recognition and has been researched extensively. See [1, 2, 5, 10, 13, 16, 17, 18] for a few examples of previous work. Unfortunately for high d most of the proposed methods are of theoretical interest only as they tend to degrade due to the dimensionality curse (a notable exception is [17]). As before, we attempt to deflate the curse by seeking an efficient algorithm that finds good approximate solutions. Let $c = d^{1/t}(4d+4)+1$ and let $q^* \in P$ denote the true nearest neighbor of an arbitrary query q under L_t. We can build a data structure in $O(d^2 n \log n)$ pre-processing time and $O(dn)$ space, so that an approximate solution $p \in P$ satisfying $dist_t(p,q) \le c{\cdot}dist_t(q^*,q)$ can be found in in $O(d^2 \log n)$ time. The basic idea of our algorithm is to locate (by binary search) the position of the query q (appropriately shifted) in each of the ZL-set lists, examine its 2 immediate neighbors (predecessor and successor) and report the closest of the up to $2(d+1)$ candidates considered. Although theoretically we can only guarantee an approximation within $O(d^{1+1/t})$ from the optimal, in practice, and for numerous data sets tested, the distance from q to the approximate solution found always differed from the true *NN* by less than 1%. Furthermore, in more than 85% of the cases the point returned was the actual *NN*. Further details on our algorithm are available in [20] (which also addresses secondary storage issues) as well as the full version of this paper.

7 Conclusions

In this paper we presented efficient approximate and exact solutions to the CP_k problem for high dimensional data. Our solutions degrade gracefully with increasing dimensions thus avoiding the exponential growth problem associated with many published algorithms.

For cases where a good approximation to CP_k suffices we presented an algorithm that runs in $O(d^2 n \log n + dk(d + \log k))$ time which finds an approximate answer set such that its k^{th} distance is within a $O(d^{1+1/t})$ factor from the true solution.

Our exact algorithm uses the approximate solution in building a candidate answer set whose size depends on the distribution of the data. When applied to data sets that satisfy certain separation conditions, our exact algorithm is particularly efficient and runs in $O(d^2 n \log n + dk)$ time in addition to the time required by the approximation phase. For general data distributions, on the other hand, no performance guarantees can be made. However, a simple process which runs in $O(d^2 n \log n)$ time can detect such cases and provide the user with the possibility of aborting the process or trying a smaller value of k. When tested against various real and synthetic data sets, our experiments have produced approximate solutions that are almost as good as the true solutions.

We are currently generalizing the approach used in this paper to solve other proximity problems for high dimensional data, including but not limited to fixed radius near neighbors, smallest k-clustering, and k-nearest neighbors.

References

[1] P.K. Agarwal and J. Matoušek. Ray shooting and parametric search. *SIAM J. Comput.*, 22:794–806, 1993.

[2] S. Arya, D. M. Mount, N. S. Netanyahu, R. Silverman, and A. Wu. An optimal algorithm for approximate nearest neighbor searching. In *Proc. 5th ACM-SIAM Sympos. Discrete Algorithms*, pages 573–582, 1994.

[3] J. L. Bentley and M. I. Shamos. Divide-and-conquer in multidimensional space. In *Proc. 8th Annu. ACM Sympos. Theory Comput.*, pages 220–230, 1976.

[4] J. L. Bentley, D. F. Stanat, and E. H. William, Jr. The complexity of finding fixed-radius near neighbors. *Inform. Process. Lett.*, 6:209–212, 1977.

[5] J. L. Bentley, B. W. Weide, and A. C. Yao. Optimal expected-time algorithms for closest-point problems. *ACM Trans. Math. Softw.*, 6:563–580, 1980.

[6] K. Beyer, J. Goldstein, R. Ramakrishnan, and U. Shaft. When is 'nearest neighbor' meaningful? In *Proc. 7th International Conference on Database Theory*, pages 217–235, 1999.

[7] B. Bhanu, J. Peng, and S. Qing. Learning feature relevance and similarity metrics in image databases. In *Proc. IEEE Workshop on Content-Based Access of Image and Video Libraries*, pages 14–18, 1998.

[8] T. Chan. Approximate nearest neighbor queries revisited. In *Proc. 13th Annu. ACM Sympos. Comput. Geom.*, pages 352–358, 1997.

[9] Bernard Chazelle. An improved algorithm for the fixed-radius neighbor problem. *Inform. Process. Lett.*, 16:193–198, 1983.

[10] K. L. Clarkson. An algorithm for approximate closest-point queries. In *Proc. 10th Annu. ACM Sympos. Comput. Geom.*, pages 160–164, 1994.

[11] A. Datta, H.-P. Lenhof, C. Schwarz, and M. Smid. Static and dynamic algorithms for k-point clustering problems. *J. Algorithms*, 19:474–503, 1995.

[12] M. T. Dickerson and R. S. Drysdale. Fixed-radius near neighbor search algorithms for points and segments. *Inform. Process. Lett.*, 35:269–273, 1990.

[13] H. Edelsbrunner, G. Haring, and D. Hilbert. Rectangular point location in d dimensions with applications. *Comput. J.*, 29:76–82, 1986.

[14] D. Eppstein and J. Erickson. Iterated nearest neighbors and finding minimal polytopes. *Discrete Comput. Geom.*, 11:321–350, 1994.

[15] C. Faloutsos and S. Roseman. Fractals for secondary key retrieval. In *Proc. ACM SIGMOD Conf. on Management of Data*, pages 247–252, 1989.

[16] J. H. Friedman, J. L. Bentley, and R. A. Finkel. An algorithm for finding best matches in logarithmic expected time. *ACM Trans. Math. Softw.*, 3:209–226, 1977.

[17] A. Gionis, P. Indyk, and R. Motwani. Similarity search in high dimensions via hashing. In *Proc. Int. Conf. on Very Large DataBases 1999*, pages 518–529, 1999.

[18] D. T. Lee and C. K. Wong. Quintary trees: a file structure for multidimensional database systems. *ACM Trans. Database Syst.*, 5:339–353, 1980.

[19] H.-P. Lenhof and M. Smid. Enumerating the k closest pairs optimally. In *Proc. 33rd Annu. IEEE Sympos. Found. Comput. Sci.*, pages 380–386, 1992.

[20] S. Liao, M. A. Lopez, and Scott Leutenegger. High dimensional similariy search with space filling curves. Tech. Report, Mathematics and Computer Science, University of Denver, 2000. In preparation.

[21] J. Salowe. L_∞ interdistance selection by parametric search. *Inform. Process. Lett.*, 30:9–14, 1989.

[22] H. Samet. The quadtree and related hierarchical data structures. *ACM Comput. Surv.*, 16(2), June 1984.

[23] C. Schwarz, M. Smid, and J. Snoeyink. An optimal algorithm for the on-line closest-pair problem. In *Proc. 8th Annu. ACM Sympos. Comput. Geom.*, pages 330–336, 1992.

[24] M. I. Shamos and D. Hoey. Closest-pair problems. In *Proc. 16th Annu. IEEE Sympos. Found. Comput. Sci.*, pages 151–162, 1975.

[25] M. Smid. Maintaining the minimal distance of a point set in less than linear time. *Algorithms Rev.*, 2:33–44, 1991.

[26] V. Turau. Fixed-radius near neighbors search. *Inform. Process. Lett.*, 39:201–203, 1991.