Closest Pair Problems in Very High Dimensions

Piotr Indyk[1], Moshe Lewenstein[2] Ohad Lipsky[2], and Ely Porat[2]

[1] MIT
indyk@theory.lcs.mit.edu
[2] Bar-Ilan University
{moshe,lipsky,porately}@cs.biu.ac.il

Abstract. The problem of finding the closest pair among a collection of points in $\Re^d$ is a well-known problem. There are better-than-naive-solutions for constant d and approximate solutions in general. We propose the first better-than-naive-solutions for the problem for large d. In particular, we present algorithms for the metrics L_1 and L_∞ with running times of $O(n^{(\omega+3)/2})$ and $O(n^{(\omega+3)/2}\log D)$ respectively, where $O(n^\omega)$ is the running time of matrix multiplication and D is the diameter of the points.

1 Introduction

The problem of finding the closest pair in a given set of points from $\Re^d$ is a fundamental and well-studied problem in computational geometry. It has been known for at least two decades [Rab76] that closest pair can be solved in $O(n)$ time (for any L_p norm) as long as the dimension d is constant. However, the complexity of this problem becomes much more mysterious when d is large. Shamos and Bentley [SB76] conjectured that, for $d = n$, the problem can be solved in $O(n^2 \log n)$ time. So far (dis)-proving this conjecture remains elusive.

In recent years, several *approximate* algorithms were designed for the high-dimensional closest pair problem [Kle97,IM98,BOR99]. In particular, the work [IM98] (cf. [Ind01]) provided a c-approximate algorithm for this problem with running time $O(dn^{1+1/c}\log^{O(1)} n)$ in L_1 and L_2 norm; recently, the time was reduced to $O(dn^{1+1/c}\log n)$ for the L_2 norm [DIIM03]. However, for the exact case of $c = 1$, the aforementioned algorithms do not provide any improvement over a naive $O(dn^2)$-time bound.

In this paper we investigate the complexity of the closest pair and related problems in very high dimensions. For simplicity, we focus on the case $d = n$. It has been known (although not published anywhere) that for the L_2 norm, the complexity of closest pair is closely related to the complexity of matrix multiplication. Specifically, one matrix multiplication enables computing dot-products of all pairs of input vectors, from which the $O(n^2)$ pairwise distances can be easily extracted in $O(n^2)$ time.

Here we show that the relationship between closest pair and matrix multiplication is not limited to the particular L_2 case. In particular, assuming that matrix multiplication can be solved in $O(n^\omega)$ time, we show:

- $O(n^{(\omega+3)/2})$-time algorithm for computing all pairwise distances according to the L_1 norm; thus the closest pair (or the diameter) can be found within the same time.
- $O(n^{(\omega+3)/2} \log D)$-time algorithm for finding closest pair under the L_∞ norm where D is the diameter of the points.
- $O(n^\omega \log n/\epsilon)$-time algorithm for $(1+\epsilon)$-approximate computing of all pairwise distances according to the L_∞ norm.

Thus, our results provide the first non-trivial bounds for the *exact* high-dimensional closest pair problem in L_1 and L_∞.

The methods we show here can be used to solve similar problems in pattern matching. For details see [LP04].

Our techniques. Our algorithms exploit the parallel between matrix multiplication and convolution of strings [Mut95], see also [MP94]. This allows us to use the "high/low-frequency" pattern matching technique of Abrahamson [Abr87] in the context of matrix multiplication. We note that while the "high/low-frequency" pattern matching technique has been used in many applications, e.g. [Kos90,DGM94,AF95,AAL,APL04], this seems to be its first application in conjunction with matrix multiplication.

2 Preliminaries and Problem Definition

We assume the RAM model of computation, which allows arithmetic on $\log N$ bit numbers in $O(1)$ time, where N is the order of the maximum problem size. Let $x = (x_1, x_2, \ldots, x_n)$ and $y = (y_1, y_2, \ldots, y_n)$ be two points in Σ^n. Then the Minkowsky L_p distance metric between x and y is defined as: $L_p(x, y) = \sqrt[p]{\Sigma_{i=1}^n |x_i - y_i|^p}$. The L_1, L_2 and L_∞ metrics are the well-known Manhattan, Euclidean and Max metrics, respectively. We deal with numerical alphabets, so $\Sigma \subset \Re$.
Notation: For a matrix X we define $X^{(b)}$ by $X^{(b)}[i,j] = (X[i,j])^b$, and X^T by $X^T[i,j] = X[j,i]$. We use ω to denote the exponent of n which is needed in order to multiply 2 matrices of size $n \times n$.

2.1 Problem definitions

- The *all pairs L_p distance* problem is defined as follows:
  Input: Set of n points $S = \{p_1, p_2, \ldots, p_n\}$ where $p_i \in \Sigma^n$.
  Output: Matrix $M[1 \ldots n, 1 \ldots n]$, where for every $i, j \in \{1, \ldots, n\}$, $M[i,j] = \|p_i - p_j\|_p$.

- The *approximate all pairs L_p distance* is defined as follows:
  Input: Set of n points $S = \{p_1, p_2, \ldots, p_n\}$, accuracy parameter $0 < \epsilon < 1$.
  Output: Matrix $\widehat{M}[1 \ldots n, 1 \ldots n]$, s.t. $M[i,j] \leq \widehat{M}[i,j] \leq (1+\epsilon)M[i,j]$, where $M[i,j] = \|p_i - p_j\|_p$.
- The *closest pair under L_p* is defined as follows:
  Input: Set of n points $S = \{p_1, p_2, \ldots, p_n\}$ where $p_i \in \Sigma^n$.
  Output: All Pairs (k,l), $1 \leq k,l \leq n$ where $\|p_k - p_l\|_p \leq \|p_i - p_j\|$ for all $i,j \in \{1, \ldots, n\}$.

For all these problems, we let $A = \{a_{i,j}\}$ denote an $n \times n$ matrix whose rows are $p_1, \ldots, p_n$, i.e. $a_{i,j} = p_i[j]$.

3 All Pairs Under L_1 Distance

Since the dimension, d, of each of the points is n it is straightforward to solve the problem of all pairs under the L_1 distance in $O(n^3)$ by directly computing the distance for each of the pairs.

In this section we provide faster algorithms for this problem. Let $A = \{a_{i,j}\}$ denote a $n \times n$ matrix whose rows are $p_1, \ldots, p_n$, i.e. $a_{i,j} = p_i[j]$. We begin by showing an $O(Sn^\omega)$ time algorithm for the problem, where $S = \max_{1 \leq j \leq n}\{\#\text{different numbers in column } j \text{ of } A\}$. We then present a different algorithm that merges the ideas of the first algorithm with an algorithm that uses the method of dividing by frequency. The running time of the second algorithm is $O(n^{(\omega+3)/2})$. Note that if $S \leq n^{\frac{3-\omega}{2}}$ then the running time of the first algorithm is better.

A central idea of both algorithms is a "divide and conquer" approach on the computation of the matrix multiplication by observing that elements of column k of A are relevant only to elements of this same column. More precisely, it is necessary to compute the matrix M, where $M[i,j] = \sum_{k=1}^{n} |a_{i,k} - a_{j,k}|$. Now, observe that for matrices X, Y,

$$X \times Y^T = \begin{pmatrix} x_{1,1} & \cdots & x_{1,n} \\ \vdots & & \vdots \\ x_{n,1} & \cdots & x_{n,n} \end{pmatrix} \times \begin{pmatrix} y_{1,1} & \cdots & y_{n,1} \\ \vdots & & \vdots \\ y_{1,n} & \cdots & y_{n,n} \end{pmatrix} =$$

$$\sum_{k=1}^{n} \begin{pmatrix} 0 \cdots 0 & x_{1,k} & 0 \cdots 0 \\ \vdots \quad \vdots & \vdots & \vdots \quad \vdots \\ 0 \cdots 0 & x_{n,k} & 0 \cdots 0 \end{pmatrix} \times \begin{pmatrix} 0 & \cdots & 0 \\ \vdots & & \vdots \\ 0 & \cdots & 0 \\ y_{1,k} & \cdots & y_{n,k} \\ 0 & \cdots & 0 \\ \vdots & & \vdots \\ 0 & \cdots & 0 \end{pmatrix}$$

From the above equation, we see that matrix multiplication of two matrices X, Y^T involves multiplying of elements of column k of X only with elements of row k of Y^T (or column k of Y). This leads us to the following algorithm which works in $O(Sn^\omega)$, where $S = \max_{j=1}^{n} |\Sigma_j|$, $\Sigma_j = \{a_{i,j} \mid 1 \le i \le n\}$.

3.1 $O(Sn^\omega)$ Algorithm

The following algorithm uses the divide and conquer approach described above and computes the necessary info for each column and value separately.

Algorithm I

1. Let $B_1, \dots, B_S, C_1, \dots, C_S, D_1, \dots, D_S, E_1, \dots, E_S$ be matrices of size $n \times n$.
2. For all $j = 1, 2, \dots, n$
   (a) For every $\sigma^j_\alpha \in \Sigma_j = \{\sigma^j_1, \sigma^j_2, \dots, \sigma^j_{|\Sigma_j|}\}$
      i. For every $i = 1, 2, \dots, n$
         A. $B_\alpha[i,j] \leftarrow 1$, if $a_{i,j} = \sigma^j_\alpha$ or $B_\alpha[i,j] \leftarrow 0$ otherwise.
         B. $C_\alpha[i,j] \leftarrow \begin{cases} a_{i,j} & a_{i,j} > \sigma^j_\alpha \\ 0 & a_{i,j} = \sigma^j_\alpha \\ -a_{i,j} & a_{i,j} < \sigma^j_\alpha \end{cases}$
         C. $D_\alpha[i,j] \leftarrow a_{i,j}$ if $a_{i,j} = \sigma^j_\alpha$ or $D_\alpha[i,j] \leftarrow 0$ otherwise.
         D. $E_\alpha[i,j] \leftarrow \begin{cases} -1 & a_{i,j} > \sigma^j_\alpha \\ 0 & a_{i,j} = \sigma^j_\alpha \\ 1 & a_{i,j} < \sigma^j_\alpha \end{cases}$
3. For every $1 \le \alpha \le S$ compute $M_\alpha = B_\alpha \times C^T_\alpha + D_\alpha \times E^T_\alpha$.
4. Compute $M = \sum_{\alpha=1}^{S} M_\alpha$.

To show that the algorithm is correct it is sufficient to show the following.

Lemma 1. *The matrix M output by the algorithm satisfies: $M[i,j] = \sum_{k=1}^{n} |a_{i,k} - a_{j,k}|$.*

Proof.

$$M[i,j] = \sum_{\alpha=1}^{S} M_\alpha[i,j] = \sum_{\alpha=1}^{S} (B_\alpha \times C^T_\alpha[i,j] + D_\alpha \times E^T_\alpha[i,j]) =$$

$$\sum_{\alpha=1}^{S} (\sum_{k=1}^{n} (B_\alpha[i,k] C_\alpha[j,k] + D_\alpha[i,k] E_\alpha[j,k])) \qquad (**)$$

Now from the way $B_\alpha, C_\alpha, D_\alpha$ and E_α are assigned we know that

$$B_\alpha[i,k] C_\alpha[j,k] + D_\alpha[i,k] E_\alpha[j,k] = \begin{cases} 0 & a_{i,k} \neq \sigma^k_\alpha \\ -a_{j,k} + a_{i,k} & \sigma^k_\alpha = a_{i,k} > a_{j,k} \\ a_{j,k} - a_{i,k} & \sigma^k_\alpha = a_{i,k} < a_{j,k} \end{cases}$$

It means that $B_\alpha[i,k]C_\alpha[j,k]+D_\alpha[i,k]E_\alpha[j,k] = \chi_=(a_{i,k},\sigma_\alpha^k)|a_{i,k}-a_{j,k}|$ (where $\chi_=(x,y)=1$ if $x=y$ and 0 otherwise). That leads to:
$(**) = \sum_{\alpha=1}^{S}(\sum_{k=1}^{n}\chi_=(a_{i,k},\sigma_\alpha^k)|a_{i,k}-a_{j,k}|) = \sum_{k=1}^{n}(\sum_{\alpha=1}^{S}\chi_=(a_{i,k},\sigma_\alpha^k)|a_{i,k}-a_{j,k}|) = \sum_{k=1}^{n}(|a_{i,k}-a_{j,k}|\sum_{\alpha=1}^{S}\chi_=(a_{i,k},\sigma_\alpha^k)) = \sum_{k=1}^{n}|a_{i,k}-a_{j,k}|$. (since $a_{i,k}$ equals to exactly one of $\sigma_1^k,\ldots,\sigma_{|\Sigma_k|}^k$). □

Note that the same result can be achieved, and even be simpler, using only half of the matrix multiplications used in the proof. However, we use this form in order to make the proofs in the $O(n^{(\omega+3)/2})$ algorithm simpler.

Time: Our time complexity is derived from the maximal number of different numbers in each row of A and from the time needed to multiply two matrices of order n. The total time needed is $O(Sn^\omega)$. For a constant S the time complexity is $O(n^\omega)$, but for large S, e.g. $S=n$, it is worse than the naive algorithm, which runs in time $O(n^3)$.

3.2 Dividing Columns of A into Groups by Frequency of Numbers

In order to reduce the time needed in case that $S > n^{\frac{3-\omega}{2}}$ we apply a technique of grouping non-frequent alphabet elements together. The process, described in detail below, divides the elements that appear in each column j of A to frequent and non-frequent numbers, then it divides the non-frequent numbers into $O(k)$ intervals, in a manner that none of the intervals contain a frequent number and at most $2n/k$ elements of the non-frequent numbers are contained in each interval.

For each column j of A, $j=1,2,\ldots,n$ do the following:

1. Define $S_j = \{< a_{ij}, i > | 1 \le i \le n\}$.
2. Sort elements of S_j by a_{ij}.
   From now on, we refer to S_j as the sorted sequence $< s_1, l_1 >, < s_2, l_2 >, \cdots, < s_n, l_n >$, i.e. $s_1 \le s_2 \le \ldots \le s_n$.
   We define a number, f, to be *frequent* if at least n/k elements in S_j have $s_i = f$. Otherwise, we say that it is *non-frequent.*
3. Divide the sequence S_j into continuous groups as follows:
   (a) For each frequent number f all elements with $s_i = f$ will form a group F_f^j.
   Now, there are at most k groups of frequent numbers, with elements of non-frequent numbers scattered into, at most, $k+1$ intervals between them. Let $I_1,\ldots,I_r$ denote these intervals ($r \le k+1$).
   (b) Divide the elements of each I_v into smaller interval-groups as follows:
   $t \leftarrow 1$
   While there are at least n/k elements in I_v that are not in groups:
   – Take the leftmost n/k elements in I_v that are not yet in any group and form a new group NF_t^j.
   – Let m be the maximum s_i s.t. $< s_i, l_i > \in NF_t^j$.

- Add to NF_t^j all the elements with $s_i = m$ (there are at most n/k, since m is non-frequent).
- $t \leftarrow t + 1$

Form a group NF_t^j of the, less than n/k, remaining elements in I_v.

At the end of this process we have for each row j at most k groups $F_{f_1}^j, F_{f_2}^j, \ldots$ because we have n elements, and the size of each F_f^j is at least n/k. In addition we have at most $k+1$ groups NF_t^j of size $< n/k$ (The groups of the "remainders"). The rest of the elements are spread over groups NF_t^j of size at least n/k. Therefore, the total number of groups is $O(k)$. From the construction of the groups it is easy to see that the following properties are satisfied.

- All the groups are intervals.
- None of the groups intersect each other. Moreover, if $< x, y >$ is in one group and $< x', y' >$ in another then $x \neq x'$.
- Each group of non-frequent numbers contains at most $2n/k$ elements.

3.3 $O(n^{(\omega+3)/2})$ Algorithm

We now describe the second algorithm. The algorithm consists of two stages. In the first stage we divide the numbers of each column into frequent numbers and intervals of non-frequent numbers as described in section 3.2. Then we apply a procedure similar to the $O(Sn^\omega)$ algorithm on the intervals. However, this will cause errors because we ignore the distance made by pairs of elements from the same interval. In the second stage we take care of correcting the errors caused by the first stage.

Algorithm Steps:
First, divide each column $j \in \{1, \ldots, n\}$ of A into groups of frequent elements $F_{f_1}^j, \ldots, F_{f_{e(j)}}^j$ $(e(j) \leq k)$, and groups of non-frequent elements $NF_1^j, \ldots, NF_{e'(j)}^j$ $(e'(j) \leq 2k+1)$, as described in section 3.2.
Define $\max(NF_c^j) = max\{x \mid < x, y > \in NF_c^j\}$, $\min(NF_c^j) = min\{x \mid < x, y > \in NF_c^j\}$, and $\min(F_f^j) = \max(F_f^j) = f$. Let $z = 3k + 1$.

First stage

1. Let $B_1, \ldots, B_z, C_1, \ldots, C_z, D_1, \ldots, D_z, E_1, \ldots, E_z$ be matrices of size $n \times n$.
2. For all $j = 1, 2, \ldots, n$
   (a) For every $G^j_\alpha \in \{G^j_1, \ldots, G^j_{e(j)+e'(j)}\} = \{F^j_{f_1}, \ldots, F^j_{f_{e(j)}}, NF^j_1, \ldots, NF^j_{e'(j)}\}$
      i. For every $i = 1, 2, \ldots, n$
         A. $B_\alpha[i,j] \leftarrow 1$, if $a_{i,j} \in G^j_\alpha$ or $B_\alpha[i,j] \leftarrow 0$ otherwise.
         B. $C_\alpha[i,j] \leftarrow \begin{cases} a_{i,j} & a_{i,j} > \max(G^j_\alpha) \\ 0 & a_{i,j} \in G^j_\alpha \\ -a_{i,j} & a_{i,j} < \min(G^j_\alpha) \end{cases}$
         C. $D_\alpha[i,j] \leftarrow a_{i,j}$ if $a_{i,j} \in G^j_\alpha$ or $D_\alpha[i,j] \leftarrow 0$ otherwise.
         D. $E_\alpha[i,j] \leftarrow \begin{cases} -1 & a_{i,j} > \max(G^j_\alpha) \\ 0 & a_{i,j} \in G^j_\alpha \\ 1 & a_{i,j} < \min(G^j_\alpha) \end{cases}$
3. For every $1 \le \alpha \le z$ compute $M_\alpha = B_\alpha \times C^T_\alpha + D_\alpha \times E^T_\alpha$.
4. Compute $M_{stage1} = \sum_{\alpha=1}^{z} M_\alpha$.

We now show the correctness of the algorithm. Beforehand, we define $\chi_{group}(x,y) = 1$ if x and y are in the same NF^k_α for some $\alpha \in \{1, \ldots, e'(k)\}$, and 0 otherwise.

Lemma 2. *The output of the first stage,* $M_{stage1}[i,j] = \sum_{k=1}^{n} |a_{i,k} - a_{j,k}| \overline{\chi_{group}(a_{i,k}, a_{j,k})}$,

Proof. The proof is similar to the proof given in section 3.1 (note, if x and y are both members of some F^k_α then it means that $x = y$). $\square$

In the second stage we compute $M_{stage2}[i,j] = \sum_{k=1}^{n} |a_{i,k} - a_{j,k}| \chi_{group}(a_{i,k}, a_{j,k})$.

Second stage: We compute M_{stage2} in a straightforward way.

For every $1 \le j \le n$
    For every $1 \le \alpha \le e'(j)$
        For every $< a, i_a >, < b, i_b > \in NF^j_\alpha$
            $M_{stage2}[i_a, i_b] \leftarrow M_{stage2}[i_a, i_b] + |a - b|$
            $M_{stage2}[i_b, i_a] \leftarrow M_{stage2}[i_b, i_a] + |a - b|$

Finally, the algorithm computes $M = M_{stage1} + M_{stage2}$, which yields $\sum_{k=1}^{n} |a_{i,k} - a_{j,k}| \overline{\chi_{group}(a_i k - a_j k)} + |a_{i,k} - a_{j,k}| \chi_{group}(a_{i,k}, a_{j,k}) = \sum_{k=1}^{n} |a_{i,k} - a_{j,k}|$ and the desired result is obtained.

Time: The first stage is implemented with $O(k)$ matrix multiplications. Therefore, the running time of the first stage is $O(kn^\omega)$. The second stage takes time $O(n \max_j(e'(j)) \max^2_{j,\alpha}(|NF^j_\alpha|)) = O(nk(\frac{2n}{k} + 1)^2) = O(n^3/k)$. Choosing $k = n^{\frac{3-\omega}{2}}$ yields an $O(n^{(\omega+3)/2})$ time for this algorithm.

4 All Pairs Under L_2 Distance

For completeness, we present here an algorithm that solves the problem of *all pairs L_2 distance* in $O(n^\omega)$. The time complexity is derived from the time needed to multiply two matrices of size $n \times n$.

By definition we need to compute $M[i,j] = \sqrt{\sum_{k=1}^{n} |a_{i,k} - a_{j,k}|^2}$ for every $i,j \in \{1,\ldots,n\}$. We show how to compute the matrix $M^{(2)}$, where $M^{(2)}[i,j] = M[i,j]^2 = \sum_{k=1}^{n} |a_{i,k} - a_{j,k}|^2$ and in one linear time pass on this matrix we compute the desired output matrix, M.

Note that $$M[i,j]^2 = \sum_{k=1}^{n} |a_{i,k} - a_{j,k}|^2 = \underbrace{\sum_{k=1}^{n} a_{i,k}^2}_{(1)} + \underbrace{\sum_{k=1}^{n} a_{j,k}^2}_{(2)} - \underbrace{2\sum_{k=1}^{n} a_{i,k}a_{j,k}}_{(3)}$$

We compute $(1), (2)$ and (3) separately, each of them for all $i,j \in 1,\ldots,n$, getting three matrices of values M_1, M_2 and M_3, summing them together gives $M^{(2)}$.
Computing M_1 and M_2 is easily done in linear time. M_3 can be computed by matrix multiplication in $O(n^\omega)$ time by observing that $M_3 = -2A \times A^T$.

5 Closest Pair Under L_∞ Distance

We define the problem of *pairs with distance of at least d with L_∞* to be: finding all pairs with a distance at least d. We solve this problem in $O(n^{(\omega+3)/2})$ using the technique of dividing the columns of A into frequent and non-frequent elements. Next, by running a binary search on d, we find the minimal d such that pairs of points with distance d exists in our set and, hence, solve the closest pair problem. (Note that there can be more than one pair of minimal distance). The time complexity is $O(n^{(\omega+3)/2} \log D)$, where $D = \max_{j=1}^{n}(\max_{i=1}^{n} a_{i,j} - \min_{i=1}^{n} a_{i,j})$ is the diameter.

5.1 Pairs with Distance At least d with L_∞

Our objective is to compute the matrix M defined by $M[i,j] = \bigvee_{k=1}^{n} \chi_d(a_{i,k}, a_{j,k})$, where $\chi_d(x,y) = 1$ if $|x - y| \geq d$ and 0 otherwise.
First, use the method described in section 3.2 in order to divide the elements of every column j of A to, at most, $3k+1$ groups, $G^j = \{G_1^j, \ldots, G_{e(j)+e'(j)}^j\} = \{F_{f_1}^j, \ldots, F_{f_{e(j)}}^j, NF_1^j, \ldots, NF_{e'(j)}^j\}$. We can assume G^j is sorted by $\max(G_\alpha^j)$ (if not, sort it). Our algorithm contains two stages, in the first stage we seek for

elements with distance $\geq d$ by their group, treating "non-sure" cases as distance $< d$ and in the second stage we check those "non-sure" cases one by one. Total time is $O(n^{(\omega+3)/2})$, independent of d.

Algorithm Outline:

Let $S_1, \ldots, S_{3k+1}, T_1, \ldots, T_{3k+1}, R_1, \ldots, R_{3k+1}$ be matrices of size $n \times n$.

First stage

1. For every $j = 1, 2, \ldots, n$
   (a) For every $G^j_\alpha \in G^j$
      i. For every $i = 1, 2, \ldots, n$
         $S_\alpha[i,j] \leftarrow 1$ if $< a_{i,j}, i > \in G^j_\alpha$ or 0 otherwise.
         $T_\alpha[i,j] \leftarrow \begin{cases} 1 \ (max(G^j_\alpha) + d \leq a_{i,j}) \vee (a_{i,j} \leq \min(G^j_\alpha) - d) \\ 0 \ otherwise \end{cases}$
      ii. Compute $R_\alpha \leftarrow S_\alpha \times T_\alpha$
   (b) Compute R defined by $R[i,j] = \bigvee_{\alpha=1}^{3k+1} R_\alpha[i,j]$.

Second stage

1. For every $j \in \{1, \ldots, n\}$
   For every $G^j_\alpha \in G^j$
      For every $< a, i_a > \in G^j_\alpha$
         For every $a_{i,j} \in$ $[\min(G^j_\alpha) - d, \max(G^j_\alpha) - d] \cup [\min(G^j_\alpha) + d, \max(G^j_\alpha) + d]$
         (a) $R[i_a, i] \leftarrow \chi_d(a_{i_a,j}, a_{i,j})$
         (b) $R[i, i_a] \leftarrow \chi_d(a_{i_a,j}, a_{i,j})$

Algorithm Correctness

Lemma 3. *At the end of the first stage* $R[i,j] = \bigvee_{k=1}^{n} (\max(a_{i,k}) + d < a_{j,k} \vee a_{j,k} < \min(a_{i,k}) - d)$, *where* $\max(a_{i,k})$ *and* $\min(a_{i,k})$ *are* $\max(G^k_\alpha)$ *and* $\min(G^k_\alpha)$ *respectively for* $a_{i,k} \in G^k_\alpha$.

Proof. Directly from the way we defined $S_1, \ldots, S_{3k+1}, T_1, \ldots, T_{3k+1}$. In other words, instead of comparing each $a_{j,k}$ with $a_{i,k}$ we only checked if $a_{j,k}$ is at least d-bigger than the maximal value of $a_{i,k}$'s group or d-smaller from the minimal value of $a_{i,k}$'s group. Now, if $\min(a_{i,k}) - d < a_{j,k} < \max(a_{i,k}) + d$ we treated it in the first stage as $|a_{i,k} - a_{j,k}| < d$. We fix it in the second stage. If we were checking in the second stage each element from G^j_α with all the elements in the interval $[\min(G^j_\alpha) - d, \max(G^j_\alpha) + d]$ we are done. But, we checked for the intervals $[\min(G^j_\alpha) - d, max(G^j_\alpha) - d]$ and $[\min(G^j_\alpha) + d, max(G^j_\alpha) + d]$. Now, we have two cases:

1. If $d < \max(G_\alpha^j) - min(G_\alpha^j)$ then $\max(G_\alpha^j) - d > min(G_\alpha^j)$ and $max(G_\alpha^j) > min(G_\alpha^j)+d$. Therefore $[\min(G_\alpha^j)-d, \max(G_\alpha^j)+d] = [\min(G_\alpha^j)-d, max(G_\alpha^j)-d] \cup [\min(G_\alpha^j)+d, max(G_\alpha^j)+d]$.
2. If $d > \max(G_\alpha^j) - min(G_\alpha^j)$ then the elements in $[max(G_\alpha^j) - d, min(G_\alpha^j)] \cup [max(G_\alpha^j), min(G_\alpha^j) + d]$ are at distance less than d from all elements of G_α^j and, therefore, do not need to be checked.

□

Time: The first stage has $O(k)$ matrix multiplications. Therefore the time is $O(kn^\omega)$. In the second stage we check all elements of each group G_α^j with all elements from the interval $[\min(G_\alpha^j) - d, max(G_\alpha^j) - d]$ ("$-d$" interval) and from $[\min(G_\alpha^j) + d, max(G_\alpha^j) + d]$ ("+d" interval). Now, since for every j the groups $G_1^j, \ldots, G_{e(j)+e'(j)}^j$ do not intersect, neither their "-d" intervals intersect nor their "+d" intervals intersect (it is possible, though that a "-d" interval of one group will intersect with a "+d" interval of another group. this fact and the fact that there are $O(n/k)$ elements in each G_α^j gives a running time of $O(\sum_{j=1}^n 2 \times O(n/k) \times n) = O(n^3/k)$. Choosing $k = n^{(3-\omega)/2}$ yields a total time of this algorithm $O(n^{(\omega+3)/2})$.

6 Approximate All Pairs Under L_∞ Distance

In this section we show how to approximate the values of $\|p_i - p_j\|_\infty$ for all $i, j \in \{1, \ldots, n\}$, up to a factor of $1+\varepsilon$ in $O(\frac{1}{\varepsilon} n^\omega \log n)$. First, we show that it is enough to compute $\|p_i - p_j\|_p$ for $p \geq \log n/\varepsilon$ in order to approximate $\|p_i - p_j\|_\infty$ up to a factor of $1+\varepsilon$. Second, we give an $O(pn^\omega)$ time algorithm to compute *all pairs L_p distance*, for even p's. Choosing an even p leads to the required results.

Lemma 4. *For every $p \geq \log n/\varepsilon$, $v, u \in \Re^n$,*

$$\|v-u\|_\infty \leq \|v-u\|_p \leq \|v-u\|_\infty(1+\varepsilon)$$

Proof: It is clear that for any p it holds that $\|v - u\|_\infty \leq \|v - u\|_p$. We need to show that $\|v - u\|_p \leq \|v - u\|_\infty(1 + \varepsilon)$. Let $v = (v_1, \ldots, v_n), u = (u_1, \ldots, u_n)$ and let $b = \max_{1 \leq i \leq n} |v_i - u_i| = \|v - u\|_\infty$. Now, $\|v - u\|_p = \sqrt[p]{\sum_{i=1}^n |v_i - u_i|^p} \leq \sqrt[p]{\sum_{i=1}^n b^p} = \sqrt[p]{nb^p} = \sqrt[p]{n}b = \sqrt[p]{n}\|v - u\|_\infty$. We have left to show that $\sqrt[p]{n} \leq (1 + \varepsilon)$ and we are done. If $p \geq \log n/\varepsilon$ then $\sqrt[p]{n} = n^{\frac{1}{p}} \leq n^{\frac{\varepsilon}{\log n}} = 2^\varepsilon \leq ((1+\varepsilon)^{\frac{1}{\varepsilon}})^\varepsilon = 1+\varepsilon$.□

6.1 All Pairs under L_p Distance for even p's

We use a technique similar to the one shown in section 4 to build this algorithm. As before, $A = \{a_{i,j}\}$ will denote a matrix whose rows are $p_1, \ldots, p_n$.

By definition we need to compute $M[i,j] = \sqrt[p]{\sum_{k=1}^{n} |a_{i,k} - a_{j,k}|^p}$ for every $i,j \in \{1,\ldots,n\}$. Note that $M[i,j]^p = \sum_{k=1}^{n} |a_{i,k} - aj,k|^p = \sum_{k=1}^{n} (a_{i,k} - aj,k)^p$ since p is even. Now, $M[i,j]^p = \sum_{k=1}^{n}(a_{i,k} - a_{j,k})^p = \sum_{k=1}^{n} \sum_{r=0}^{p} (-1)^r \binom{p}{r} a_{i,k}^r a_{j,k}^{p-r} = \sum_{r=0}^{p} \sum_{k=1}^{n} (-1)^r \binom{p}{r} a_{i,k}^r a_{j,k}^{p-r} = \sum_{r=0}^{p} (-1)^r \binom{p}{r} \sum_{k=1}^{n} a_{i,k}^r a_{j,k}^{p-r}$ (*).

The computation of M is done as follows:

1. Compute $A^{(r)}$ for every $0 \leq r \leq p$.
2. Compute $B_r = A^{(r)} \times (A^{(p-r)})^T$ for $0 \leq r \leq p$ (T denote transpose).
3. Compute $M = \sum_{r=0}^{p} (-1)^r \binom{p}{r} B_r$.
4. For every $i,j \in \{1,\cdots,n\}, M[i,j] \leftarrow \sqrt[p]{M[i,j]}$.

It is clear from equation (*) that the computation is correct.

Time: The time needed for the first and third steps of the computation is $O(pn^2)$ and the time needed for the second step is $O(pn^\omega)$. Therefore, the total time of the algorithm is $O(pn^\omega)$.

References

[Abr87] K. Abrahamson. Generalized string matching. *SIAM J. Computing*, 16(6):1039–1051, 1987.

[AAL] A. Amir, A. Apostolico, and M. Lewenstein. Inverse Pattern Matching. *J. of Algorithms*, 24(2):325-339, 1997.

[AF95] A. Amir and M. Farach. Efficient 2-dimensional approximate matching of half-rectangular figures. *Information and Computation*, 118(1):1-11, 1995.

[APL04] A. Amir, E. Porat and M. Lewenstein. Faster algorithms for string matching with k mismatches. *J. of Algorithms, special SODA 2000 issue*, to appear.

[BOR99] A. Borodin, R. Ostrovsky, and Y. Rabani. Subquadratic approximation algorithms for clustering problems in high dimensional spaces. *Proceedings of the Symposium on Theory of Computing*, 1999.

[DIIM03] M. Datar, N. Immorlica, P. Indyk, and V. Mirrokni. Locality-sensitive hashing scheme based on p-stable distributions. *Proc. of Symposium of Computational Geometry (SOCG)*, 2004, to appear.

[DGM94] M. Dubiner, Z. Galil, and E. Magen. Faster tree pattern matching. *J. of the ACM*, 41(2):205-213, 1994.

[IM98] P. Indyk and R. Motwani. Approximate nearest neighbor: towards removing the curse of dimensionality. *Proceedings of the Symposium on Theory of Computing*, 1998.

[Ind01] P. Indyk. *High-dimensional computational geometry*. Ph.D. Thesis, Department of Computer Science, Stanford University, 2001.

[Kos90] R. Kosaraju. Efficient tree pattern matching. *Proc. of Symosium on Foundation of Computer Science*, 1990.

[Kle97] J. Kleinberg. Two algorithms for nearest-neighbor search in high dimensions. *Proceedings of the Twenty-Ninth Annual ACM Symposium on Theory of Computing*, 1997.

[LP04] O. Lipsky and E. Porat. Approximated Pattern Matching with the L_1 L_2 and L_∞ Metrics. Manuscript, submitted to *SODA 2004*.

[Mut95] S. Muthukrishnan. New results and open problems related to non-standard stringology. *CPM*, 1995.

[MP94] S. Muthukrishnan and K. Palem. Non-standard stringology: algorithms and complexity. *Proc. of the ACM Symposium on Theory of Computing*, 770-779, 1994.

[Rab76] M. O. Rabin. Probabilistic algorithms. *Algorithms and Complexity, J. F. Traub, editor. Academic Press*, pages 21–39, 1976.

[SB76] I. Shamos and J. Bentley. Divide-and-conquer in multidimensional space. *Proceedings of the Symposium on Theory of Computing*, pages 220–230, 1976.