In this lecture, we will give an algorithm which, given a point, finds its k approximate nearest neighbors. We then discuss how we can use an approximate nearest neighbors algorithm to find an approximate closest pair among n points.

1 Finding k approximate nearest neighbors

Given q, let $p_{i_1}, p_{i_2}, \ldots, p_{i_k}$ be the ordered actual k nearest neighbors of q in P. We would like to return $p'_{i_1}, p'_{i_2}, \ldots, p'_{i_k}$ such that for all $j = 1, \ldots, k$,

$$d(q, p'_{i_j}) \leq (1+\epsilon) d(q, p_{i_j})$$

We would like the query time to be poly(d)log(n).

Recall from last time the idea of distinguishing sets: Let V be a distinguishing set. For any $x, y \in \mathbb{R}^d$ such that $\|x\|(1+\gamma) \leq \|y\|$, x has a smaller inner product with a majority of the vectors in V.

Fact: A random set of $O(d \log^2 d) = L$ vectors suffices for V. Note that $d = \frac{1}{\gamma^2}$.

The idea behind the algorithm is that instead of comparing $\|p_i - q\|$ and $\|p_j - q\|$, we instead compare $\|(p_i - q) \cdot v_l\|$ and $\|(p_j - q) \cdot v_l\|$ for $v_l \in V$.

We note that $\|(p_i - q) \cdot v_l\| = \|p_i \cdot v_l - q \cdot v_l\|$. Thus, we can compute $p_i \cdot v_l$ beforehand.

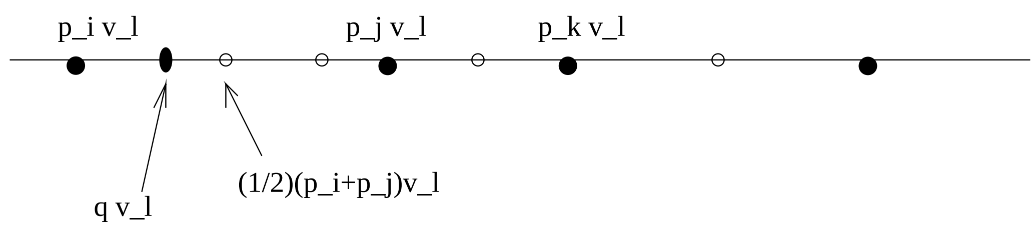

Figure 1: Projection onto v_l

In the pre-processing, for each $v_l \in V$, $l = 1, \ldots, L$, we can compute all $p_i \cdot v_l$, all $\frac{1}{2}(p_i + p_j) \cdot v_l$, and sort them all.

The set of all projections onto v_l, when excised, break the line into *primitive intervals.*

When a query shows up, we will compute the projection and figure out which primitive interals it defines.

Goal: Show that the collection of primitive intervals in which q's projection lies is enough to determind a correct set of k approximate nearest neighbors.

Definition 1. *The trace of q is a sequence of primitive intervals $\sigma_1, \sigma_2, \ldots, \sigma_L = \vec{\sigma}$ in which q's projection lies.*

A trace σ gives a majority-vote outcome for each decision "Is $d(p_i, q) \leq d(p_j, q)$?"

We can encode this as a directed complete graph (a "tournament") G_σ.

How do we then fine an approximate nearest neighbor by looking at G_σ?

One possibility here would be to analyze the cycles in G_σ. However, it is not clear how to make such an approach work. Instead, we will consider an *apex*.

Definition 2. *An apex of a tournament is a node with a path of length ≤ 2 to all other nodes.*

Lemma 3. *Every tournament has an apex.*

Proof. Look at a node v of maximum outdegree. Let V' be the set of nodes such that there is a (directed) edge from v. Suppose there is no path of length ≤ 2 to u. Then there is an edge from u to v and also from u to all nodes in V'. Thus, u has higher outdegree. Contradiction!

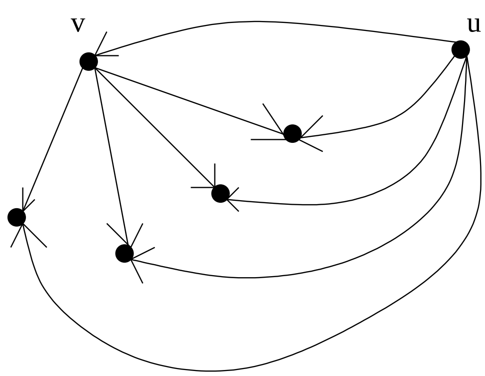

Figure 2:

□

We can define an *apex ordering* of a tournament by sequentially extracting apexes.

Claim 4. *For any k, if q has trace σ, then the first k nodes in an apex ordering of G_σ form an approximate set of k nearest neighbors where V can distinguish to within $1 + \frac{\epsilon}{3}$.*

Proof. Let us first consider the case where $k = 1$.

Let p be an apex of G_σ, p^* be the true nearest neighbor of q.

Thus, p has a path of length ≤ 2 to p^*.

Suppose it has the length-2 path $p \to p' \to p^*$. Then $d(p, q) \leq (1 + \frac{\epsilon}{3}) d(p', q)$ and $d(p', q) \leq (1 + \frac{\epsilon}{3}) d(p^*, q)$. Thus, $d(p, q) \leq (1 + \frac{\epsilon}{3})^2 d(p^*, q) \leq (1 + \epsilon) d(p^*, q)$.

For $k > 1$, either the returned value p'_{i_k} is among the true k nearest neighbors, or some point p^* among the true k nearest neighbors follow it in apex ordering. Then use $k = 1$ argument. □

Our algorithm therefore looks as follows:

Preprocess:

Enumerate all traces.

For each σ:

Construct G_σ.

Store an "apex ordering".

If done in a naive way, this will store $O(n^2)^L$ orderings (where each ordering has n values). This can be reduced by using various geometrical arguments to show that certain traces are not possible, but we will not cover these here.

The query part will take the form:

Query:

For $l = 1, 2, \ldots, L$:

Compute $q.v_l$.

Binary search $q.v_l$ through the primitive intervals corresponding to v_l.

Loop gives the trace σ. Read first k values in the apex ordering corresponing to σ.

Computing $q.v_l$ takes $O(d)$ work, and the binary search takes $O(\log n)$. The loop is of length $L = O(d \log^2 d)$ and so the overall running time of the query will therefore be

$$O(d^2 \log^2 d + d \log^2 d \log n + k)$$

where we include the $O(k)$ work involved in reading the k values in the apex ordering as k may be much larger than the other two elements.

Note that this algorithm can be viewed as a partitioning of $\mathbb{R}^d$. As in Figure 3, we are given a set of points. Each vector in the distinguishing set corresponds to a set of parallel lines in the diagram, namely the lines through the midpoints and the lines through the points themselves. Thus, when a new query arrives it will fall in a region, like the grey shaded one. These correspond to traces and each has an ordering attatched to it.

We now consider trying to improve our algorithm by means of dimension-reduction. This will rely on the following surprising result, which we quote without proof.

The Johnson-Lindenstrauss Lemma

Let P be a set of n points in $\mathbb{R}^d$. Choose a random d'-dimensional subspace S (for example by choosing d' orthonormal vectors), where $d' = \frac{9}{\epsilon^2} \log n$ and project P onto S. Then, with high probability, the ratio of distances between points in P in S will be within a $(1+\epsilon)$-factor of their distances in $\mathbb{R}^d$.

Thus a possible improvement would be to carry out such a projection and run our algorithm on the space of reduced dimension. We must beware, however, as if $d = \frac{c}{\epsilon^2} \log n$ and

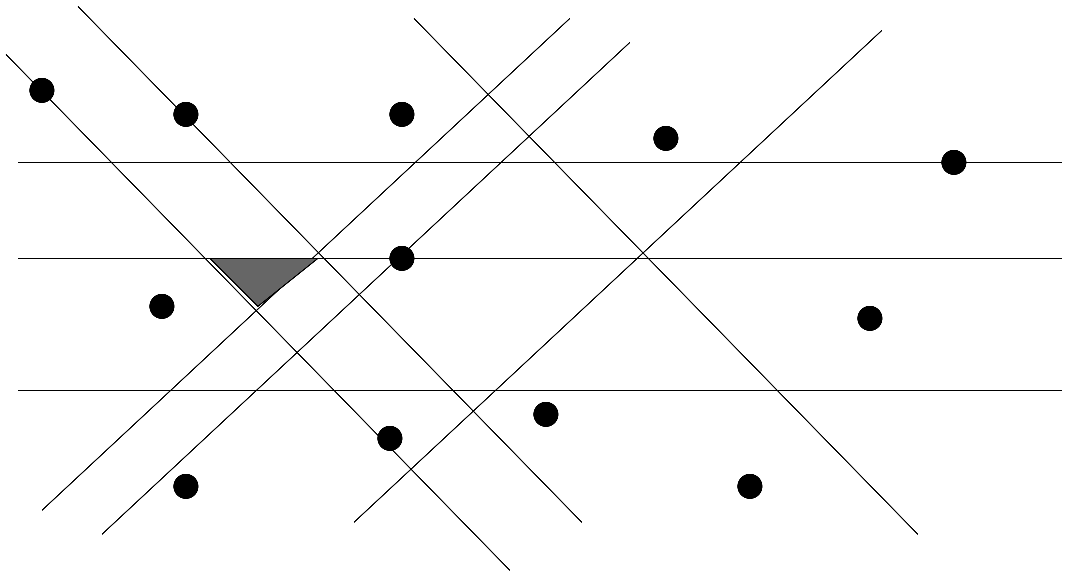

Figure 3: Is our algorithm like space partitioning?

our query processing takes $2^d \log n$ then, substituting, this is $n^{\frac{c}{\epsilon^2}\log n} > n$ and so is worse than brute force. So in fact it is not clear that the Johnson-Lindenstrauss Lemma can really be applied in a useful way to the algorithm we have just discussed. Note, in fact, that the nearest-neighbor algorithm above uses random projections not for dimension-reduction but in fact for a kind of *dimension-expansion* — it maps the d coordinates of each point to $\Theta(d \log^2 d)$ projected values, from which the trace is enough to recover a set of approximate nearest neighbors.

The table below summarizes recent developments in algorithms to solve this problem.

Algorithm	Preprocess time	Query time
Ours	$O(n^2)^L$	$O(\text{poly}(d)\text{poly}(\log n))$
Kushilevitz-Ostrovsky-Rabani (98)	$n^{O(1/\epsilon^2)}$	$O(\text{poly}(d)\text{poly}(\log n))$
Indyk-Motwani (98)	$n^{O(1/\epsilon^2)}$	$O(\text{poly}(d)\text{poly}(\log n))$
Open Question	$O(n^2)$	$O(\text{poly}(d)\text{poly}(\log n))$

The Kushilevitz-Ostrovsky-Rabani algorithm is quite similar to the one we have just discussed; however, it extracts richer information from the primitive intervals than just which of two points a query is closer to (such as how far away it is from both, etc.). The Indyk-Motwani algorithm is based on dimension reduction.

We now give an application of these two algorithms with $n^{O(1/\epsilon^2)}$ preprocessing time to the approximate closest pair problem; given n points in $\mathbb{R}^d$ which two are (approximately) closest? To make things specific we suppose the preprocessing takes n^{c/ϵ^2} time.

A naive algorithm is to project randomly into $\mathbb{R}^{(c/\epsilon^2)\log n}$ and then compute all pairwise distances. This has running time $O(n^2 \log n)$. It turns out to be quite hard to improve on this.

In 1999, Borodin, Ostrovsky and Rabani gave an algorithm that does, with running time $O(n^{2-\frac{\epsilon^2}{c}}\text{poly}(\log n))$.

In this algorithm, we first split up the n points (arbitrarily) into groups of size $n^{\epsilon^2/c}$ and preprocess each for nearest neighbour queries. The time per group is $(n^{\epsilon^2/c})^{c/\epsilon^2} = n$ and there are $n^{1-\epsilon^2/c}$ groups and so this step takes $n^{2-\epsilon^2/c}$ time.

Then, for each point p, we use the point as a query itself to find its approximate nearest neighbour in each group and then find the minimum over these. For a particular point the nearest neighbour search will take poly$(\log n)$ time per group. Thus, recalling that there are $n^{1-\epsilon^2/c}$ groups and we must do this for each of the n we see that this part of the algorithm also takes $n^{2-\epsilon^2/c}\text{poly}(\log n)$ time.