Algorithms II November 24 and 25, 2025

Lecture 19-20: Locality Sensitive Hashing

Notes by Ola Svensson[1]

Today we are going to cover a cool topic called locality sensitive hashing with the useful application of nearest neighbor search.

These notes on are basically the lecture notes of Lecture 6 in Shayan Oveis Gharan's course "CSE 521: Design and Analysis of Algorithms I" available here:

http://courses.cs.washington.edu/courses/cse521/17wi/

We start with a reduction of the nearest neighbor search (NNS) problem to that of finding a locality sensitive hashing function as invented in [**IM98**].

1 Introduction to the Nearest Neighbor Search Problem

The NNS problem is as follows: Suppose $P \subset \mathbb{R}^d$ is a set of n points. Given any $q \in \mathbb{R}^d$ find

$$\min_{p \in P} \text{dist}(p, q).$$

The distance here could be any arbitrary distance function; in this lecture we will talk more about ℓ_1 or ℓ_2 distances even though the machinery that we describe can be generalized to a variety of distance functions. Some applications include: web search, document search, or clustering - these are all situations in which knowing how "far" an object is from other objects tells us important information.

A naive solution would be to store all of the points and simply loop over all $p \in P$ to find the minimum distance. This takes $O(n \cdot d)$ time and space, which is not good. Ideally we would like to have a query time that is sublinear in n; we may allow for a super-linear amount of memory to store the data structure.

If $d = 1$ we could pre-process the points by sorting them and then finding the distance minimizing point would simply reduce to binary searching for p in a list, and returning the closest of the two adjacent elements in the list. This takes $O(\log n)$ query time and $O(n)$ bits of memory.

Extending the pre-processing idea to higher dimensions d leads to what are known as k-d trees: here the idea is to partition the space by using coordinate-aligned planes chosen appropriately for the data at hand. Unfortunately k-d trees generally fail to beat the naive approach when $d = \Omega(\log n)$. It turns out that in all known approaches the size of the data structure (or the query-time) grows exponentially in d.

The main underlying difficulty is the well-known facts in high dimensions, which is usually referred to as the "curse of dimensionality". Suppose we partition the space by a grid where each cell is a cube of side length a. Then, a cube of side length a randomly positioned in the space intersects 2^d many cells of the grid. This phenomenon essentially implies that a NNS algorithm based on kd-trees takes time $O(2^d)$ in expectation to look into all of the nearby cells of a query point to find the closes point.

2 Reducing to Approximate Nearest Neighbors Search

We now describe the idea of [**IM98**]. Firstly, instead of solving the exact problem we will look for approximate solutions. That is instead of finding the closest point p to a query point q, we are happy to find a point $p \in P$ such that

$$\text{dist}(p, q) \leq c \cdot \min_{s \in P} \text{dist}(s, q),$$

[1] **Disclaimer:** These notes were written as notes for the lecturer. They have not been peer-reviewed and may contain inconsistent notation, typos, and omit citations of relevant works.

where $c > 1$ is the approximation factor of in our algorithm. As we will see the memory and the query time of our algorithm will be a function of c.

So, let us define the approximate NNS problem. For $c > 1, r > 0$, the ANNS(c, r) is defined as follows: Given a set point of points P, construct a data structure such that for any query point q, if there is a point p such that $\text{dist}(p, q) \leq r$, it returns a point p' such that

$$\text{dist}(p', q) \leq c \cdot r.$$

It is not hard to see that we can give a c approximation to the nearest neighbor search problem using the solution to ANNS(c, r). In fact, all we need to do is to guess $\min_{p \in P} \text{dist}(p, q)$ up to a multiplicative factor of $1 \pm \epsilon$. By an appropriate scaling assume

$$\text{diam}(P) = \max_{p, p' \in P} \text{dist}(p, p') \leq 1$$

Also, suppose $\delta > 0$ is the minimum possible distance for all pairs of points in our dataset. Roughly speaking, $1/\delta$ can represent the bit precision of the data points stored in our system. We solve ANNS$(c(1-\epsilon), r)$ for the following values of r,

$$\delta, (1+\epsilon)\delta, (1+\epsilon)^2\delta, \ldots, 1.$$

We report the minimal value of r for which we find a point at distance $c(1-\epsilon)$ of q. This reduction imposes an additional $O(\log \frac{1}{\delta})$ overhead to the query time and the memory of our algorithm. This is because we need to maintain a separate data structure for each possible value of r in the above sequence.

3 Locality Sensitive Hashing functions

From now on we only focus on the ANNS(c, r). The main interesting idea of [**IM98**] is a reduction from this problem to the design of a locality sensitive hash (LSH) function. Roughly speaking, an LSH is a hash function which is sensitive to distance. Ideally, we would like to have a hash function that maps "close points" to the same value with a high probability and maps "far points" to different values. To be more precise, if $\text{dist}(p, q) \leq r$ we want them to map to the same value, with a high probability, and if $\text{dist}(p, q) > c \cdot r$ we want them to map to different values with a high probability. Let us give a formal definition.

Let U be the universe that contains the points P. Examples of U are $\mathbb{R}^d$ and all binary vectors $\{0, 1\}^d$ with d coordinates. Suppose we have a family a functions $\mathcal{H} = \{h \colon U \to \mathbb{Z}\}$ of maps from the universe U to the set of integer $\mathbb{Z}$; we say $\mathcal{H}$ is $(r, c \cdot r, p_1, p_2)$-LSH if: for all $p, q \in U$:

$$\begin{aligned} \text{dist}(p, q) \leqslant r &\implies \mathbb{P}\left[h(p) = h(q)\right] \geqslant p_1 \\ \text{dist}(p, q) \geqslant c \cdot r &\implies \mathbb{P}\left[h(p) = h(q)\right] \leqslant p_2 \end{aligned}$$

where the probabilities are over $h \sim \mathcal{H}$. Ideally, we want to have $p_1 \gg p_2$, but as we see this highly depends on the magnitude of c. The main idea in the reduction of [**IM98**] is that even if p_1 is slightly larger than p_2 it is possible to use many independently chosen functions from $\mathcal{H}$ to *boost* p_1 to a number close to 1 and p_2 to $1/n$.

Before describing the reduction, let us give an example of LSH for binary vectors. We will see more examples in the exercise session and homework. Suppose $P \subseteq \{0, 1\}^d$ with Manhattan distance function

$$\text{dist}(p, q) = \|p - q\|_1,$$

i.e., $\text{dist}(p, q)$ is the number of coordinates at which p and q have different bits. Consider the family $\mathcal{H} := \{h_i\}_{i=1}^d$ where

$$h_i(p) = p_i$$

is the ith bit of p. Then observe that for each $p, q \in \{0,1\}^d$

$$\mathbb{P}\left[h(p) = h(q)\right] = \frac{\#\text{ bits in common}}{\text{total bits}} = \frac{d - \|p-q\|_1}{d} = 1 - \frac{\|p-q\|_1}{d}.$$

Therefore,

$$\mathbb{P}\left[h(p) = h(q)\right] = \begin{cases} \geq 1 - \frac{r}{d} \approx e^{-r/d} & \text{if } \operatorname{dist}(p,q) \leq r \\ \leq 1 - \frac{c \cdot r}{d} \approx e^{-c \cdot r/d} & \text{if } \operatorname{dist}(p,q) \geq c \cdot r \end{cases}.$$

So, $\mathcal{H}$ is $(r, c \cdot r, e^{-r/d}, e^{-c \cdot r/d})$-LSH.

4 Reduction to LSH

Now let us discuss the reduction from ANNS(c, r) to LSH. Well if we had a $(r, c \cdot r, p_1, p_2)$-LSH family such that $p_1 \approx 1$ and $p_2 \approx 0$ we could solve the problem as follows: We start by choosing a function $h \sim \mathcal{H}$ uniformly at random and we store $h(p)$ for all points in P. Given a query point q, we compute $h(q)$ and see if there is any point $p \in P$ where $h(p) = h(q)$. Note that we can do the lookup in $O(1)$ time using a hash table as we discussed in previous lectures. Now, first consider a point p such that $\operatorname{dist}(p, r) \leq r$. In that case we have $h(p) = h(r)$ with probability $p_1 \approx 1$ and so the algorithm finds p. On the other hand, any point p such that $\operatorname{dist}(p, r) > c \cdot r$ satisfies $h(p) = h(r)$ with probability $p_2 \approx 0$ and so we will have few (close to none) unwanted collisions.

Thus, at least intuitively, we only need to show that if we are given an $(r, c \cdot r, p_1, p_2)$-LSH family with the assumption $p_1 > p_2$, then we can boost it to get $p_1 \approx 1$ and $p_2 \approx 0$.

We do this boosting in two steps. First, we just try to make p_2 small. To do this it suffices to take k independent hash functions from $\mathcal{H}$, and hash each point $p \in P$ to a k-dimensional vector,

$$h(p) = [h_1(p), \ldots, h_k(p)].$$

Then, by the independence of $h_1, \ldots, h_k$, for any two points p, q,

$$\operatorname{dist}(p, q) \geqslant c \cdot r \implies \mathbb{P}\left[h(p) = h(q)\right] \leqslant p_2^k.$$

But this doesn't help us increase p_1. In fact, the above hash function maps two close points to the same vector with probability at least p_1^k. How do we do this? We choose ℓ independent copies of the above k-dimensional hash function, $f_1, f_2, \ldots, f_\ell$, for a sufficiently large ℓ, with high probability there is an i such that $f_i(p) = f_i(q)$. Assume,

$$\begin{aligned} f_1(p) &= [h_{1,1}(p), \ldots, h_{1,k}(p)] \\ &\vdots \\ f_\ell(p) &= [h_{\ell,1}(p), \ldots, h_{\ell,k}(p)] \end{aligned}$$

It follows that if $\operatorname{dist}(p, q) \leq r$, then

$$\begin{aligned} \mathbb{P}\left[\exists i \mid f_i(p) = f_i(q)\right] &= 1 - \mathbb{P}\left[\forall i, f_i(p) \neq f_i(q)\right] \\ &= 1 - \mathbb{P}\left[f_i(p) \neq f_i(q)\right]^\ell \\ &\geqslant 1 - (1 - p_1^k)^\ell \end{aligned}$$

The details of the algorithm is described in Algorithm 1.

Next, we describe how to tune the parameters k, ℓ. We choose k such that $p_2^k = 1/n$. Also, assume

$$p_1 = p_2^\rho, \tag{1}$$

for some $\rho < 1$. As we will see ρ is the main parameter that determines the running time/memory of our algorithm. We choose $\ell \propto n^\rho \ln n$.

Fix a query point q; it follows by linearity of expectation that for any i,

$$\mathbb{E}\left[\#p : \mathrm{dist}(p,q) > c \cdot r, f_i(p) = f_i(q)\right] \leq n \cdot p_2^k \leq 1.$$

Summing up over all i, in expectation there are $O(\ell)$ points in our data set which map to the same hash value as q for some i. This implies an overhead of $O(\ell)$ in the query time.

On the other hand, if $\mathrm{dist}(p,q) \leq r$ for some $p \in P$, then

$$\mathbb{P}\left[\exists i : f_i(p) = f_i(q)\right] \geq 1 - (1 - p_1^k)^\ell = 1 - (1 - p_2^{\rho k})^\ell = 1 - (1 - n^{-\rho})^\ell \approx 1 - e^{-\ell n^{-\rho}} = 1 - 1/n.$$

In summary, for any point p at distance at most r, our algorithm outputs p with probability at least $1 - 1/n$. The algorithm in expectation had $O(\ell \cdot d)$ overhead to examine $O(\ell)$ points at distance more than $c \cdot r$ from q.

Algorithm 1 LSH Algorithm

Preprocessing:

Choose $k \cdot \ell$, $h_{1,1}, \ldots, h_{\ell,k}$ functions uniformly at random from $\mathcal{H}$.

Construct ℓ hash tables; for all $1 \leq i \leq \ell$ store $f_i(p) = (h_{i,1}(p), \ldots, h_{i,k}(p))$ for all $p \in P$ in the i-th hash table.

Query(q):

for $i = 1 \to \ell$ **do**

Compute $f_i(q)$.

Go over all points p where $f_i(p) = f_i(q)$. As soon as we find a point p with $\mathrm{dist}(p,q) \leq c \cdot r$, return and output p.

end for

We remark that, as we are only interested in solving the ANNS(c, r) problem (and not the problem of finding all close points), the above algorithm stops after finding a single element of distance at most $c \cdot r$, i.e., after inspecting in expectation $O(\ell)$ elements in total.

5 Space and Time Complexity of the Reduction

The algorithm needs to maintain $O(\ell)$ hash tables. In each hash table we need to store $n = |P|$ hash values where each value is a k dimensional vector. So, the space complexity of the algorithm is

$$O(\ell \cdot n \cdot k) = O(n^{1+\rho} \ln n \frac{\log n}{\log(1/p_2)}),$$

since $\ell = O(n^\rho \ln n)$ and $k = O(\frac{\log n}{\log(1/p_2)})$.

We now analyze the query time. For any query point q we need to spend $O(\ell \cdot k)$ time to compute $f_i(q)$ for all $1 \leq i \leq \ell$. For any candidate close point p we spend $O(d)$ time to calculate $\mathrm{dist}(p,q)$. In expectation we examine $O(\ell)$ far points that we do not want to output (points p such that $\mathrm{dist}(p,q) > c \cdot r$). So, the total query time (for computing hash functions and inspecting $O(\ell)$ far points) is

$$O(\ell \cdot k + d \cdot \ell) = O(n^\rho \ln(n)(\frac{\log n}{\log(1/p_2)} + d)).$$

Ignoring lower order terms, the algorithm runs with memory $O(n^{1+\rho})$ and querytime $O(n^\rho)$.

Let us calculate ρ for the binary vector example that we described at the beginning. Recall that ρ is chosen such that $p_2^\rho = p_1$, so

$$\rho = \frac{\ln \frac{1}{p_1}}{\ln \frac{1}{p_2}} = \frac{r/d}{c \cdot r/d} = \frac{1}{c}.$$

For example, if $c = 2$, we need $O(n^{1.5})$ to store hash tables and we have $O(\sqrt{n})$ query time. As we see the query time (and memory) get significantly better as we increase c. In practice, we may tune the parameter c based on the amount of resources available to us.

It has been a very active area of research to design the best of LSH functions for many metrics. In the exercise session and homework, we design LSH functions for some distances.