CSC473: Advanced Algorithm Design Winter 2018

Week 3: Approximate Near Neighbor Search

Aleksandar Nikolov

1 Introduction

Last week we discussed the randomized $O(n)$ expected time algorithm to compute the closest pair of points in the plane. This week we continue with a related data structure problem: (approximate) near neighbor search. Suppose you have a database D of n entries: images, text documents, census data, etc. Using a Dictionary data structure, for example a B-tree or a hash table, you can check if an entry x is in D. But sometimes you actually want to find an entry y in the database which is as "close" as possible to x, and not necessarily exactly the same. This is one of the most basic ways to do classification, for example: you keep a database of labeled images, and given a new image you want to see if it is close to any image in the database; if your new image is close to an image of a dog, then maybe it is also an image of a dog. Or, more simply, databases can contain errors, and looking for an approximate match is a way to make our database search more robust.

How is this related to the closest pair of points problem? The idea is that we can think of the entries of the database D as elements of a metric space. I.e. each database entry x is an element of some *universe* X, and we have a *distance function* d which assigns a non-negative distance to every pair of points. Usually we insist that d is a *metric*, which means that it is

1. *Reflexive*: $d(x, y) = 0$ if and only if $x = y$;
2. *Symmetric*: $d(x, y) = d(y, x)$ for all $x, y \in X$;
3. Satisfies the *Triangle Inequality*: $d(x, y) \le d(x, z) + d(z, y)$ for all $x, y, z \in X$.

Then the pair (X, d) is called a *metric space*. One very basic metric is given by the Euclidean distance between x and y, where x, y are points in the plane, or 3-dimensional space, or in general d-dimensional space. In this language, the closest pair of points problem is to find two distinct points x, y, in an input set $P \subseteq X$, that minimize the distance $d(x, y)$. In *nearest neighbor search*, our goal is design a data structure that maintains a set (database) D of points (from X). We need to be able to insert and delete points from the data structure. Most importantly, given a *query point* x, we want to efficiently output a point $y \in D$ which has the smallest distance $d(x, y)$ to x.

2 Approximate Near Neighbors for Hamming Distance

2.1 Setup

Let us move away from the abstract definition of metric spaces, and look at something very concrete. Suppose you have a database D consisting of n strings of d bits each. I.e. D is a subset of $\{0, 1\}^d$. For a string x, we write x_i for its i-th bit (an alternative notation would be to think of x as an array of d bits, and write $x[i]$). In the nearest neighbor problem we want to design a data structure that maintains D in memory and supports the following operations:

- Insert(D, x): insert x into D;
- NearestNeighbor(D, x): output the closest string y to x in D.

Here closest means the string y that minimizes the Hamming distance

$$d_H(x, y) = |\{i : x_i \neq y_i\}|,$$

i.e. the number of bits in which x and y differ.

This data structure problem turns out to be very hard. For all known data structures that solve it, at least one of the two operations runs in time $\Omega(\min\{2^d, dn\})$. In other words, when d is on the order of $\log n$, they are no better than just storing a list of all strings in D, and doing a linear search to find the nearest neighbor. (This is one example of the *curse of dimensionality.*)

Exercise 1. *Give a data structure for which* Insert *can be implemented in time $O(1)$, and* NearestNeighbor *can be implemented in time $O(2^d)$.*

To overcome this challenge, we relax the requirements a little bit. Instead of NearestNeighbor, we ask for another procedure:

- ApxNearestNeighbor(D, x): output a string $y \in D$ such that

  $$\min\{d_H(x, z) : z \in D\} \leq d_H(x, y) \leq C \cdot \min\{d_H(x, z) : z \in D\}, \tag{1}$$

  where $C > 1$ is a constant.

I.e. we do not insist that we find the exact nearest neighbor, and instead, we are satisfied to find a string which is not much further from x than the nearest neighbor. The constant C is called the *approximation factor.* This is the *approximate nearest neighbor search* problem (ANNS). We call a string y that satisfies (1) a C-near neighbor of x.

Exercise 2. *Suppose there exists a data structure for which* Insert *and* ApxNearestNeighbor *run in time $T(n)$. Use these two operations to find a pair x, y of distinct strings in a given set D in time $O(T(n)n)$ such that*

$$d_H(x, y) \leq C \min\{d(x', y') : x', y' \in D, x' \neq y'\}.$$

Using a clever hashing scheme (due to Indyk and Motwani), we will give a randomized data structure for this problem. In this data structure, Insert and ApxNearestNeighbor will run in time $O(dn^\rho)$, where $\rho \approx \frac{1}{C}$, and also ApxNearestNeighbor(D, x) will output a C-near neighbor of x with probability at least $\frac{2}{3}$. It will be enough to implement another function NearNeighbor$_r(D, x)$, with the following guarantees:

1. If there exists a string z in D such that $d_H(x, z) \leq r$, then NearNeighbor$_r(D, x)$ will output some $y \in D$ for which $d_H(x, y) \leq Cr$;

2. If every string z in D satisfies $d_H(x,z) \geq Cr$, then $\textsc{NearNeighbor}_r(D,x)$ outputs $\textsc{Fail}$.

Exercise 3. *Assume that we can implement* $\textsc{NearNeighbor}_r(D,x)$ *for* $r \in \{0, 1, \sqrt{C}, \sqrt{C}^2, \ldots, C^{\lceil \log_{\sqrt{C}} d \rceil}\}$, *so that for each r in this set it runs in time* $T(n)$*, satisfies property 1. with probability at least* $\frac{2}{3}$*, and property 2. with probability* 1*. Show that we can then implement* $\textsc{ApxNearestNeighbor}(D,x)$ *with approximation factor* C *to run in time* $O(T(n)\log_C(d))$ *and success probability* $\frac{2}{3}$.

2.2 Buckets

Recall that in the data structure we used to solve the closest pair of points problem we discretized our space and then, when we need to find a point q close to a query point p, we just used a hash table to check if any cell around p is non-empty. We could try something similar here. Our space is already discrete, so we can skip the discretization step. Then, given a query string x we can use a hash table to check if D contains any string in the set $\{y : d_H(x,y) \leq r\}$. However, the size of this set is

$$\binom{d}{0} + \binom{d}{1} + \ldots + \binom{d}{r},$$

which, for $r = \frac{d}{2}$ is 2^{d-1}, and can be much larger than n if $d \geq 10 \log_2 n$, for example. This is, once again, the curse of dimensionality.

To avoid this problem, we use *two-level* hashing. The first level is the key idea of the data structure. We will define a random hash function g with the property that for any two strings x and y such that $d_H(x,y) \leq r$, the probability that $g(x) = g(y)$ is large, and for any two strings such that $d_H(x,y) \geq Cr$, the probability that $g(x) = g(y)$ is small. Think of $g(x)$ as defining a bucket in which x falls. We expect the near neighbors of x to fall in the same bucket as x, and the strings that are far from x to fall in a different bucket. Then the near neighbors of x can be found using ordinary hashing.

For a multiset $I = \{i_1, \ldots, i_k\}$ of indexes, let $g_I(x) = x_{i_1}x_{i_2}\ldots x_{i_k}$. I.e., g just maps x to the k-bit string consisting of the coordinates of x indexed by I. We pick a random hash function g_I by picking k random indexes in $\{1, \ldots, d\}$, uniformly and independently with replacement.

Lemma 1. *For the random hash function* g_I *described above, and any two d-bit strings* $x, y \in \{0,1\}^d$*, we have*

$$d_H(x,y) \leq r \implies \mathbb{P}(g_I(x) = g_I(y)) \geq p_1^k, \tag{2}$$

$$d_H(x,y) \geq Cr \implies \mathbb{P}(g_I(x) = g_I(y)) \leq p_2^k, \tag{3}$$

where $p_1 = 1 - \frac{r}{d}$ *and* $p_2 = 1 - \frac{Cr}{d}$.

Proof. Let i be a uniformly random index in $\{1, \ldots, d\}$. For any two d-bit strings x and y,

$$\mathbb{P}(x_i = y_i) = \frac{|\{i : x_i = y_i\}|}{d} = 1 - \frac{|\{i : x_i \neq y_i\}|}{d} = 1 - \frac{d_H(x,y)}{d}.$$

Then, if $d_H(x,y) \leq r$, we have that $\mathbb{P}(x_i = y_i) \geq p_1$; if $d_H(x,y) \geq Cr$, we have that $\mathbb{P}(x_i = y_i) \leq p_2$. The lemma follows immediately because

$$\mathbb{P}(g_I(x) = g_I(y)) = \mathbb{P}(x_{i_1} = y_{i_1} \text{ and } \ldots \text{ and } x_{i_k} = y_{i_k}) = \mathbb{P}(x_i = y_i)^k.$$

The final equality is due to the fact that the indexes $i_1, \ldots, i_k$ are chosen uniformly and independently. $\square$

We call all strings x in D with the same hash value $g_I(x)$ a *bucket*. The value $g_I(x)$ itself is the *label* of the bucket. Thus, we have the property that strings that are near each other are more likely to end up in the same bucket than strings that are far.

2.3 The Two-Level Hashing Scheme

Our data structure consists of L hash tables $T_1, \ldots, T_L$, each with m slots, $m \geq n$. We will assume that the hash function $h_\ell : \{0,1\}^k \to \{1, \ldots, m\}$ is used with the hash table T_ℓ. Our requirement for h_ℓ is the standard one: that the probability of any two distinct elements colliding is bounded by $\frac{1}{m}$. We will resolve collisions by chaining, i.e. every slot $T_\ell[i]$ of each hash table is a pointer to the head of a linked list. We will also keep L hash function $g_{I_1}, \ldots, g_{I_L}$, defined as in the previous section, where I_ℓ is a multiset of k indexes from $\{1, \ldots, d\}$, chosen independently and uniformly with replacement.

The Insert procedure is straightforward:

```
INSERT($D, x$)
1  for $\ell = 1$ to $L$
2      Insert $x$ at the head of the linked list $T[h_\ell(g_{I_\ell})]$
```

The worst-case running time of Insert, assuming that evaluating h_ℓ takes time $O(k)$, is $O(kL)$.

The NearNeighbor$_r$ procedure is not much more complicated:

```
NEARNEIGHBOR$_r$($D, x$)
 1  num-checked = 0
 2  for $\ell = 1$ to $L$
 3      $i = h_\ell(g_{I_\ell}(x))$
 4      Set $y$ to the head of $T_\ell[i]$
 5      while $y \neq$ NIL
 6          if $d_H(x, y) \leq Cr$
 7              return $y$
 8          num-checked = num-checked + 1
 9          if num-checked == $12L + 1$
10              return FAIL
11          else Set $y$ to the next element in $T_\ell[i]$
12  return FAIL
```

In words, the procedure inspects the cells that the bucket $g_{I_\ell}(x)$ hashes to, starting from $\ell = 1$ to $\ell = L$. For each ℓ, it inspects the linked list in the hash table cell, and checks whether any of the strings stored in it is at distance at most Cr from x. Once the procedure has exhausted all L hash table cells, or has inspected at least $12L + 1$ strings, it quits and outputs Fail. Because it takes

$O(k)$ time to compute each hash value, and $O(d)$ time to compute the Hamming distance between two strings, the worst-case running time of $\textsc{NearNeighbor}_r$ is $O((k+d)L)$.

Before we analyze the data structure, we need a couple of basic facts from probability theory. One of them is the *union bound*: for any two events A and B in a probability space, $\mathbb{P}(A \text{ or } B) \le \mathbb{P}(A) + \mathbb{P}(B)$. This is obvious: draw a Venn diagram of A and B and convince yourself that

$$\mathbb{P}(A \text{ or } B) = \mathbb{P}(A) + \mathbb{P}(B) - \mathbb{P}(A \text{ and } B) \le \mathbb{P}(A) + \mathbb{P}(B),$$

because probabilities are non-negative. Using a simple induction, this inequality extends to k events: if $A_1, \ldots, A_k$ are events in a probability space, then

$$\mathbb{P}(A_1 \text{ or } \ldots \text{ or } A_k) \le \mathbb{P}(A_1) + \ldots + \mathbb{P}(A_k).$$

Note that we do not need to know anything at all about the events $A_1, \ldots, A_k$: they do not need to be independent or disjoint, for example. To quote one of my math professors, the union bound is a very powerful triviality.

The other fact we need is Markov's inequality.

Lemma 2 (Markov's Inequality)**.** *Let $X \ge 0$ be a random variable. Then, for any $x > 0$,*

$$\mathbb{P}(X > x) < \frac{\mathbb{E}[X]}{x}.$$

Proof. By the law of total expectation:

$$\mathbb{E}[X] = \mathbb{E}[X \mid X \le x]\mathbb{P}(X \le x) + \mathbb{E}[X \mid X > x]\mathbb{P}(X > x).$$

Because $X \ge 0$, the first term is non-negative. The second term is strictly bigger than $x\mathbb{P}(X > x)$. Therefore

$$\mathbb{E}[X] > x\mathbb{P}(X > x),$$

and we get the inequality by re-arranging. $\square$

Both these simple facts are *very important* and will be used many times in the course.

We are now ready to analyze the data structure. The definition of the $\textsc{NearNeighbor}_r$ guarantees that it will never output a string with Hamming distance more than Cr from x. Our main goal is to bound the probability that the procedure outputs $\textsc{Fail}$ if there actual is a string at distance at most r from x. This is done in the following theorem.

Theorem 3. *Let p_1 and p_2 be as in Lemma 1 and let $k = \lceil \log_{1/p_2}(n) \rceil$. Let $\rho = \frac{\log(1/p_1)}{\log(1/p_2)}$, and let $L = 2n^\rho$. If there exists a string z^* in D such that $d_H(x, z^*) \le r$ then, with probability at least $\frac{2}{3}$, $\textsc{NearNeighbor}_r(D, x)$ will output some $y \in D$ for which $d_H(x, y) \le Cr$.*

Proof. Let us fix some $z^* \in D$ such that $d_H(x, z^*) \le r$; such a string exists by assumption. Let $F = \{z \in D : d_H(x, z) > Cr\}$. Let us say that the procedure $\textsc{NearNeighbor}_r(D, x)$ succeeds if it outputs some $y \in D$ for which $d_H(x, y) \le Cr$. In order for this to happen, it is clearly enough that the following two properties hold

1. a string in F collides with x at most $2L$ times;

2. z^* collides with x.

Above "collide" is used in the following sense: a string y collides with x if $h_\ell(g_{I_\ell}(y)) = h_\ell(g_{I_\ell}(x)$ for some ℓ. We count collisions with multiplicity: if some $y \in F$ collides with x in different hash tables T_ℓ, we count each collision separately. We will show that both properties hold simultaneously with constant probability.

An element $y \in F$ can collide with x in T_ℓ in one of two cases: either $u = g_{I_\ell}(x) \neq g_{I_\ell}(y) = v$ but $h_\ell(u) = h_\ell(v)$, or $g_{I_\ell}(x) = g_{I_\ell}(y)$. Let us fix an arbitrary $y \in F$, and consider these two cases separately. In the first case, recall that we picked each hash table to have $m \geq n$ slots, and assumed that h_ℓ is chosen so that $\mathbb{P}(h_\ell(u) = h_\ell(v)) \leq \frac{1}{m} \leq \frac{1}{n}$ for any $u \neq v$. In the second case, by Lemma 1 and our choice of k we have that $\mathbb{P}(g_{I_\ell}(x) = g_{I_\ell}(y)) \leq p_2^k \leq \frac{1}{n}$. Since the two cases are disjoint, we have that

$$\mathbb{P}(y \text{ collides with } x \text{ in } T_\ell) \leq \frac{1}{n} + \frac{1}{n} = \frac{2}{n}.$$

Let $X_{y,\ell}$ be the indicator random variable which equals 1 if y collides with x in T_ℓ, and equals 0 otherwise. Let $X = \sum_{y \in F} \sum_{\ell=1}^{L} X_{y,\ell}$ be the total number of collisions with x. By linearity of expectation,

$$\mathbb{E}[X] = \sum_{y \in F} \sum_{\ell=1}^{L} \mathbb{P}(X_{y,\ell} = 1) \leq \frac{2|F|L}{n} \leq 2L,$$

where we used the fact that $|F| \leq |D| = n$. By Markov's inequality, $\mathbb{P}(X > 12L) < \frac{1}{6}$. This establishes that the first property holds with probability at least $\frac{5}{6}$.

Let us now estimate the probability that z^* collides with x. This probability is at least as large as the probability that there exists an $\ell \in \{1, \ldots, L\}$ for which $g_{I_\ell}(x) = g_{I_\ell}(z^*)$. By Lemma 1, and because the different I_ℓ were chosen independently,

$$\begin{aligned} \mathbb{P}(\exists \ell : g_{I_\ell}(x) = g_{I_\ell}(z^*)) &= 1 - \prod_{\ell=1}^{L} (1 - \mathbb{P}(g_{I_\ell}(x) = g_{I_\ell}(z^*))) \\ &\geq 1 - (1 - p_1^k)^L \geq 1 - e^{-Lp_1^k}. \end{aligned}$$

To simplify the calculation, assume that $k = \log_{1/p_1}(n)$. Then

$$p_1^k = 2^{-k \log_2(1/p_1)} = 2^{-\frac{\log_2(n)}{\log_2(1/p_2)} \log_2(1/p_1)} = n^{-\rho}.$$

Therefore, $Lp_1^k = 2$, and we have that z^* collides with x with probability at least $1 - \frac{1}{e^2}$. By the union bound, the probability that both properties hold is at least $1 - (\frac{1}{6} + \frac{1}{e^2}) > \frac{2}{3}$. This finishes the proof. $\square$

Theorem 3 implies that INSERT and NEARNEIGHBOR$_r$ run in time $O((k + d)n^\rho)$. Using the approximation $1 - x \approx e^{-x}$, which is accurate for x close to 0, we see that $\rho = \frac{\log(1/p_1)}{\log(1/p_2)} \approx \frac{1}{C}$, and $k = O(d \log n)$. So, overall, both operations run in time $O(dn^{1/C} \log n)$. Importantly, for $C > 1$ this is much faster than the time $\Theta(dn)$ taken by simple linear search.

If we want NEARNEIGHBOR$_r$ to succeed with probability at least $1 - \delta$, it is enough to run it independently $O(\log(1/\delta))$ times, and check if any of the runs succeeded. (**Exercise: do the calculation.**)

3 Locality Sensitive Hashing

Let us now see how we can adapt what we just did for Hamming distance to other distance functions. The key idea was to have a hash function that is more likely to put nearby points in the same bucket, than to put far away points in the same bucket. The precise definition follows.

Definition 4. *Let $d : X \times X \to \mathbb{R}$ be a distance measure defined on the universe X. A random hash function h with domain X is locality sensitive with parameter ρ if*

$$\begin{aligned} d(x,y) \le r &\implies \mathbb{P}(h(x) = h(y)) \ge p_1, \\ d(x,y) \ge Cr &\implies \mathbb{P}(h(x) = h(y)) \le p_2, \end{aligned}$$

and $\rho = \frac{\log(1/p_1)}{\log(1/p_2)}$.

Lemma 1 shows that the hash function $h(x) = x_i$, where i is a random index in $\{1, \ldots, d\}$, is locality sensitive for Hamming distance.

Exercise 4. *Check that we can implement* INSERT *and* NEARNEIGHBOR$_r$ *for the distance function d with the same running time and guarantees as in the previous section by substituting the hash function g_I with g defined by $g(x) = (h_1(x), \ldots, h_k(x))$, where $h_1, \ldots, h_k$ are independently chosen locality sensitive hash functions.*

This approach gives approximate near neighbor search algorithms for many important distance functions, for example Euclidean distance in high dimension.

Exercise 5. *The Manhattan, or ℓ_1, distance between two d-dimensional points $x, y \in \{0, \ldots, N-1\}^d$ is defined by*

$$d_1(x,y) = \sum_{i=1}^{d} |x_i - y_i|.$$

For example, in $d = 2$, this distance tells us how many steps we need to go from x to y, if we are only allowed to go up and down, and left and right. Give a locality sensitive hash function for Manhattan distance with the same p_1 and p_2 as in Lemma 1. The function should be computable in time $O(1)$.

The survey by Andoni and Indyk [AI08] has more information about locality sensitive hashing. The website `http://www.mit.edu/~andoni/LSH` contains links to some of the latest papers, and two different implementations.

References

[AI08] Alexandr Andoni and Piotr Indyk. Near-optimal hashing algorithms for approximate nearest neighbor in high dimensions. *Commun. ACM*, 51(1):117–122, 2008.