CS 493: Algorithms for Massive Data Sets **Approximate Nearest Neighbor Searching**
04/09/2002 **Scribe: Ding Liu**

1 Review

1.1 Locality Sensitive Hash Functions

Given a collection $\mathcal{C}$ of objects and a similarity function $sim(x, y)$, a locality sensitive hash function family $\mathcal{F}$ operates on $\mathcal{C}$, such that for any $x, y \in \mathcal{C}$,

$$\text{Prob}_{h \in \mathcal{F}}[h(x) = h(y)] = sim(x, y)$$

In the last lecture we showed that if $sim(x, y)$ admits a locality sensitive hash function family, then the distance function $1 - sim(x, y)$ satisfies the triangle inequality.

1.2 The Approximate Nearest Neighbor Searching Problem

In the last lecture we also considered the approximate nearest neighbor searching problem in Hamming space. The problem is as follows. Given a collection $\mathcal{P}$ of n points in the Hamming cube H^d (that is, each point is a 0-1 vector in dimension d). We want to preprocess $\mathcal{P}$ such that given any query point q, we can quickly find an $(1 + \epsilon)$-approximate nearest neighbor of q in $\mathcal{P}$. We call p^* an $(1 + \epsilon)$-approximate nearest neighbor of q if for any $p \in \mathcal{P}$, $H(p^*, q) \leq (1 + \epsilon)H(p, q)$. Here $H(x, y)$ is the Hamming distance between x and y.

2 Finding Approximate Nearest Neighbor via Locality Sensitive Hash Functions

In this section we will solve the $(1+\epsilon)$-approximate nearest neighbor problem using locality sensitive hash functions. In the last lecture we introduced the (r_1, r_2)-neighbor problem. There we gave an overview of the data structure of the (r_1, r_2)-neighbor problem without analyzing it. In this lecture we will finish its analysis. But before that, we'd like to show how to use it in the approximate nearest neighbor problem.

Recall that we can regard the (r_1, r_2)-neighbor problem as a black box that distinguishes between the following two cases:

- If there exists a point p whose distance from the query is at most r_1, then it returns a point p' whose distance from the query is at most r_2.
- If all points are at distance more than r_2 from the query, then it answers "no".

To solve the $(1+\epsilon)$-approximate nearest neighbor problem, we build several data structures for the (r_1, r_2)-neighbor problem with different values of (r_1, r_2). More specifically, we could explore (r_1, r_2) equal to $(r_{min}, r_{min}(1+\epsilon)$, $(r_{min}(1+\epsilon), r_{min}(1+\epsilon)^2$, ..., $r_{max}(1+\epsilon)^{-1}, r_{max}$, where r_{min} and r_{max} are the smallest and the largest possible distance between the query and the data point, respectively. We run our algorithm (to be described) on the above values of (r_1, r_2), and stop when the algorithm says "yes" for the first time. The point p' returned by the algorithm is outputted as the $(1+\epsilon)$-approximate nearest neighbor. If each call of the (r_1, r_2)-neighbor problem is answered correctly, then p' is indeed a $(1+\epsilon)$-approximate nearest neighbor of the query. To see this, look at the first call (r_1, r_2) when "yes" is returned. We know that all points are at distance more than r_1 from the query since the last call answered "no" (the last call was made on $(r_1/(1+\epsilon), r_1)$). Also the point p' returned by the current call is at distance at most $r_2 = (1+\epsilon)r_1$ from the query, this means that p' is a $(1+\epsilon)$-approximate nearest neighbor of the query. One little issue left. Since the algorithm for the (r_1, r_2)-neighbor problem is randomized, there is some error probability. We deal with this issue by making the error probability small enough. We then bound the error probability of the $(1+\epsilon)$-approximate nearest neighbor algorithm by the union bound.

Now we are at the point to analyze the algorithm for the (r_1, r_2)-neighbor problem. We adopt notations from the last lecture. Recall that we formed l groups, with each group defined by k locality sensitive hash functions. We let $g_j = (h_{j1}, \cdots, h_{jk})$ be the k hash functions for the j-th group. Recall that we compute $g_1(q), \ldots, g_l(q)$ and test those p such that p agrees with q in at least one group. We interrupt the search after finding $2l$ points, including duplicates. From the last lecture we know that the algorithm is correct if the following two properties hold:

1. If there exists p such that $H(p, q) \leq r_1$ then $g_j(p) = g_j(q)$ for some $j = 1, \ldots, l$.
2. The total number of p such that $H(p, q) > r_2$ and yet $g_j(p) = g_j(q)$ for some $j = 1, \ldots, l$ is less than $2l$.

Recall that for any hash function h used in the above construction, $\text{Prob}[h(p) = h(q)] = sim(p, q) = 1 - H(p, q)/d$. Thus for a point p such that $H(p, q) \leq r_1$, $\text{Prob}[h(p) = h(q)] \geq p_1$ where $p_1 = 1 - r_1/d$. On the other hand, $H(p, q) > r_2$ implies that $\text{Prob}[h(p) = h(q)] < p_2$ where $p_2 = 1 - r_2/d$. So for any g_j, $H(p, q) \leq r_1$ implies that $\text{Prob}[g_j(p) = g_j(q)] \geq p_1^k$; $H(p, q) > r_2$ implies that $\text{Prob}[g_j(p) = g_j(q)] < p_2^k$. Now we set $k = \log n / \log(1/p_2)$ so that $p_2^k = 1/n$. Since there are at most n candidate p such that $H(p, q) > r_2$, the expected number of p for which $g_j(p) = g_j(q)$ hold (for a fixed g_j) is at most 1. Then the expected number of p such that $H(p, q) > r_2$ yet agrees with q in *any* g_j is at most l. By Markov inequality the probability that this number exceeds $2l$ is less than $1/2$. This shows that the second property listed above holds with probability at least $1/2$.

How about the first property? When $H(p, q) \leq r_1$, for any fixed g_j, $\text{Prob}[g_j(p) = g_j(q)] \geq p_1^k = p_1^{\log n / \log(1/p_2)} = n^{-\rho}$ where $\rho = \log(1/p_1)/\log(1/p_2)$. Thus the probability that p agrees with q in at least one group is at least $1 - (1 - n^{-\rho})^l$. Setting $l = n^\rho$ makes this probability

lower bounded by $1 - 1/e$. This is the probability for property one to hold. By the union bound we know that with probability at least $1/2 - 1/e$, both these two properties hold.[1]

Now we try to bound the time and space of this algorithm. We first bound k and l. $k = \log n / \log(1/p_2) = O(\log n)$. To bound l we need to bound ρ first. Recall that $\rho = \log(1/p_1)/\log(1/p_2)$ and $p_1 = 1 - r_1/d$, $p_2 = 1 - r_2/d$. We may let $r_2 = (1+\epsilon)r_1$ since this is always the case each time we call this algorithm. It is not hard to show that $\rho = O(1/(1+\epsilon))$. We omit the proof here, which could be found in [2]. Thus $l = O(n^{1/(1+\epsilon)})$. Thus the space of this algorithm is $O(dn + nlk) = \tilde{O}(dn + n^{1+1/(1+\epsilon)})$, and the query time is $O(lkd) = \tilde{O}(dn^{1/(1+\epsilon)})$.[2] For example, if we set $\epsilon = 1$ then we can find a 2-approximate nearest neighbor in time $\tilde{O}(d\sqrt{n})$, using space $\tilde{O}(n(d+\sqrt{n}))$.

3 Another Algorithm for Approximate Nearest Neighbor Searching

The algorithm in the last section requires separate data structures for different possible values of r, the distance between the query and its nearest neighbor. In this section we give another algorithm from [1] which automatically adjusts to the correct distance r. The time and space of this new algorithm is the same as the old one.

Given n 0-1 vectors in dimension d, we choose $N = O(n^{1/(1+\epsilon)})$ random permutations of the bits. For each random permutation σ, we maintain a sorted order O_σ of the n vectors, in lexicographic order of the bits permuted by σ. Given a query q we do the following: For each σ we perform a binary search on O_σ to locate the two vectors closest to q. We then search in each O_σ, examining vectors above and below the position returned by the binary search in order of the length of the longest prefix that matches q. This is done by maintaining two pointers for each sorted order O_σ (one moves up and the other down). At each step we move one of the pointers up or down corresponding to the element with the longest matching prefix. We examine $2N = O(n^{1/(1+\epsilon)})$ vectors in this way. Of all the vectors examined, we return the one that has the smallest Hamming distance to q.

The analysis of this algorithm is very similar to the one in the last section. Suppose the nearest neighbor of q is at a Hamming distance of r from q. Let $p_1 = 1 - r/d$, $p_2 = 1 - r(1+\epsilon)/d$, $k = \log_{1/p_2} n$, $\rho = \log(1/p_1)/\log(1/p_2)$. Then $n^\rho = O(n^{1/(1+\epsilon)})$. We can show that with constant probability the following two conditions hold:

1. From amongst $O(n^{1/(1+\epsilon)})$ permutations, there exists a permutation σ such that the nearest neighbor agrees with q on the first k coordinates in σ.

[1] By repeating the above algorithm $O(\log(1/\delta))$ times, we can amplify the probability of success in at least one trial to $1 - \delta$, for any $\delta > 0$. For example, the $(1+\epsilon)$-approximate nearest neighbor problem calls this algorithm $\log_{(1+\epsilon)} d$ times, we then let $\delta < 1/\log_{(1+\epsilon)} d$, so that the algorithm for the $(1+\epsilon)$-approximate nearest neighbor problem succeeds with constant probability. For this, we repeat the (r_1, r_2)-neighbor algorithm $O(\log\log d)$ times.

[2] $\tilde{O}(f(n,d))$ represents $O(f(n,d))\log^c n$ for some constant c.

2. Over all N permutations, the number of vectors that are at Hamming distance of more than $r(1+\epsilon)$ from q and agree on the first k coordinates is less than $2N$.

The correctness of this algorithm follows immediately from these two conditions.

A nice property of this data structure is that we do not need a reduction to many instances of the (r_1, r_2)-neighbor problem. That is, we solve the nearest neighbor problem simultaneously for all values of distance r using a single data structure.

References

[1] Charikar, M. *Similarity estimation techniques from rounding algorithms*, Proc. ACM STOC (2002), To appear.

[2] Indyk, P., Motwani, R. *Approximate nearest neighbors: Towards removing the curse of dimensionality*, Proc. ACM STOC (1998), 604–613.