Optimal Data-Dependent Hashing for Approximate Near Neighbors: final report

Sungil Ahn, Almas Abdibayev

1 Introduction and motivation

In the Nearest Neighbour problem, we are given a set P of n points in d-dimensional space, and are asked to return the closest point to an arbitrary query point q. For low d, there are algorithms to provide the exact solution efficiently (e.g. Voronoi diagram for $d = 2$). However, as the dimensions increase, those algorithms break down, and the cost of distance computation becomes increasingly costly.

To address this, we can relax our requirements and ask for a point that is "close enough": in the Approximate Near Neighbour problem (ANN), we are asked to provide a point that is within cr distance (where c is our approximation factor) from q if there exists a data point within distance r.

This approach is useful for studying the underlying structure of data - while not exactly similar to clustering in terms of its goals or globality of its span, ANN does help us to study and analyze each datapoint in the context of other datapoints, which is especially helpful in comparison of high dimensional vector representations of said data. For instance, due to the speed of ANN, we could search through billions of images to match images, or study the properties of the generative models by sampling two closely related vectors that produce different "fake" datapoints.

In fact, the data-dependent hashing of this paper makes it somewhat akin to clustering, as both iteratively adapt to the shape of the dataset, and both essentially "cluster" objects (which, in the context of our paper, correspond to partitions of the state space, as explained later).

Interestingly, the algorithm presented in the paper adapts to the structure of the dataset despite having no prior knowledge of the dataset itself nor assuming any structure of the points, and results in better performance and insight than the non-adaptable algorithms.

Another theme (more thought is given to this in critiques section) touched on by this paper is the embedding high dimensional vectors R^d into objects of with certain properties is being explored in the realm of word embeddings. One example is the embedding of entities in knowledge bases (in the form of the word vectors) into hyperbolic space. The hyperbolic space naturally provides us with soft (non-discrete and continuous) approximation of a tree data structure, which is omnipresent among relational knowledge bases. One can thus modify the standard training procedure for word embeddings to take into account relations between words and approximate the structure present within the data explicitly (as opposed to embedding vectors in Euclidian space which does not have any natural way to represent trees, which would force us to come up with separate algorithm to reveal such structures). [NK17] This paper, however, has a fixed way to embed the objects onto a sphere without any optimization involved. Why is any of this important? As highlighted before we are interested in adapting to data with no prior knowledge of its structure. It is not completely true that we have no prior knowledge however. The choice of embedding structure itself is a form prior knowledge in the form of a heuristic that we as computer scientists choose: objects that have "nice" properties. Some of these objects are "nicer" than others, e.g. more generalizable than others to all sorts of distributions of data. Thus, the meta question of such choice enters our field of vision as we read this paper.

2 Basics

Having mentioned interesting aspects of this work, we can continue with the explanation of the paper, starting with the basics of the underlying method.

Recall that in class, to address ANN, we introduced the Locality-Sensitive Hash family (LSH), which had the desirable property where if the points were close, they would collide with high probability p_1, and if they were far apart, they would not collide with high probability p_2. This property was used to determine that if the hash of the query collided with some point, then there was a high probability that the point is close to the query.

We proved that this scheme took $O(n^{(1+\rho)})$ space and preprocessing time, with $O(dn^\rho)$ query time, with $\rho = \frac{\ln p1}{\ln p2} = \frac{1}{c}$ as the measure of quality for the algorithm. While this is a far cry from the $O(dn)$ query time of the naïve approach to ANN, subsequent research [DIIM04] showed that $\rho \leq \frac{1}{c^2}$, and research on the lower bound showed that this bound is tight [MNP07].

So is $\rho = \frac{1}{c^2}$ the best we can do? Using regular LSH, yes. However, let us consider what the LSH family is really doing. While drawing a hash function at random exhibits the locality-sensitive property, each hash function is simply mapping the state space according to some predetermined rule. So if P happens to be "dense" in one place but sparse everywhere else, the hash function does not make use of the information, and might even hash points from the dense part away from each other, destroying the locality.

Thus, data-dependent hashing – a randomized hash family that depends on the actual points in P - can help solve ANN more efficiently. The intuition is as follows: as the goal of the hash functions is to "clump up" points that are close together and separate distant points, we can essentially treat them as ways to partition space; so instead of partitioning space based on predefined rules, we can partition iteratively based on how dense/sparse certain parts of the state space are in order to get a more "optimal" partition.

For instance, [AINR14] was able to obtain $\rho = \frac{\frac{7}{8}}{c^2} + \frac{O(1)}{c^3}$ using data-dependent LSH. However, for large values of c, preprocessing took quadratic time, and as the iterative partitioning always started with data-independent LSH, it inevitably led to inefficiencies.

This paper presents an improvement over the previous approach, with $\rho = \frac{1}{2c^2-1}$ (which turns out to be optimal [MNP07]). In addition, it effectively reduces ANN on an arbitrary dataset to ANN on an essentially random dataset, and provides a near-linear preprocessing time!

3 General idea

So how does this work?

The basic idea is to place every data point on a sphere, and to carve out "dense clusters" into caps so we can have smaller enclosing spheres, after which the "pseudo-random" remainder will be left. For the dense clusters, we can enclose them in slightly smaller balls, and recurse. For the remainder, we can split that space, and recurse on every partition – and the smaller the radius of the enclosing ball, the more sensitive the function is to locality, and the better the ρ.

This iterative partitioning forms something akin to a decision tree, and we can query a point by "following down" the paths to the leaves. And, to boost our probability of success, we can make $O(n^{\rho})$ copies of the above spherical LSH structure, just as we have in class to boost probability of success with data-independent LSH.

But why are we working with spherical LSH in particular? Just like every other LSH, it provides high chance of colliding for close points and low chance for distant points. However, it turns out that if the condition $\|u - w\|, \|v - w\| \in \sqrt{2} \pm \epsilon, \|u - v\| \leq 1.99$ is met, the event of u colliding with w becomes almost independent of u colliding with v. This "three point property" will come into play later.

While the paper shows proof of this property (Theorem 3.1), it does not provide any good intuition as to why placing points on a sphere rather than the universe would provide the benefits, not to mention there is no discussion of how one would arrive to this conclusion nor discussion of potential benefits of choosing different and perhaps, more complex geometrical objects as a hyperparameter of this algorithm.

4 High level description of algorithm

Anyway, with this intuition in mind, we can construct the concrete steps for the algorithm:

For starters, we previously assumed that we're working exclusively on a ball. To reduce the general space R^d down to the ball, just like before, we still look for clusters (i.e. balls of radius $2c^2$) that contain at least τm points. However, the difference here is that since we're taking the intersect of R^d and a ball, the resulting "cap" is the ball itself. Other than that, we can recurse on it like any other cap. Additionally, to partition the remainder, we want an LSH that operates on R^d rather than a sphere, so we use [DIIM04] for that purpose.

Additionally, when we think about how we "carve out" clusters, we are essentially intersecting the state space with some ball and enclosing it with a sphere. However, that means most of the clusters are not on the surface of the sphere, but rather inside it. To deal with this, we must somehow reduce the query on the cap down to the query on a sphere.

To do this, we can "discretize" the ball by rounding all data points' distances to o (to a multiple of δ, which provides the trade-off between "accuracy" and "speed"), and since the query can be arbitrary, we round all possible distances to o as well so that we can just look up the rounded distance from q to o during query time. Then, for each possible pair of distances δi (from o to p) and δj (from o to q), we can create new encompassing spheres by only taking all data points that have the same "rounded" distance (δi). Note that the resulting sphere would have projections of $r_1 + 2\delta$ and $r_2 - 2\delta$ since a. we are changing distances by up to 2δ by rounding to the nearest multiple of δ ($\because$ triangle inequality), and b. the rounded position of q might not be on the sphere formed by the sphere centering at o with radius δi. The intuitive explanation for this is that for each point within the ball we draw a sphere that passes through it and then create a second sphere through which a potential query point will pass. We then outline every possible outer sphere for different query points, bounded by certain constraints. Once we have this two spheres separated by a certain distance, we can project the potential query point lying on outer sphere onto inner sphere, "hashing" it into bucket defined by a sector outlined on the sphere. Then we recurse on resulting inner sphere.

While it seems like the rounding and projection would introduce errors, as is shown in the paper, the errors can be controlled by carefully setting the parameters. However, such a convoluted scheme still makes us wonder if there is a way to somehow round positions of p and q in a way that makes q the center rather than o, which would allow bypassing the projection step at the very least.

In regards to spheres of clusters, we note that for some values of $r1, r2$ and R, we already know how to solve the ANN. For instance, when $r_2 \geq 2R$, literally any point on the sphere would satisfy the query. Or, if $r_2 \geq \sqrt{2}R$, we can directly the apply the spherical LSH family to the data points P and get the bound we want ($\because$ Thm. 2.3). Similarly, with $\frac{r_1}{r_2} \leq \frac{1}{(2c^2-1)}$, we can directly apply [DIIM04] to P and get our ρ-bound of $\frac{1}{2c^2-1}$.

As we still have a decision tree-esque structure like before, we can query a point by querying each dense cluster-enclosing ball and any nonzero $P \cap R(q)$ recursively. And at each ball "node", we can see whether there is any point satisfying the query by "rounding" q so that the distance between o and q is a multiple of δ (just like what we did while preprocessing), then for all possible potential (rounded) distances between p and o, in a similar manner to the preprocessing step, we can query the corresponding sphere. The query eventually stops when one of the spheres fall into one of the 3 cases in which we can partition the sphere with guarantee that ρ has the bound we want.

5 High level analysis of the algorithm

We encourage the interested reader to peruse the original paper for detailed analysis. Here we present condensed version of it.

5.1 Convergence

The general idea is that for dense clusters, we are reducing the radius of the enclosing sphere with each iteration, and for the pseudorandom remainders, the points more or less lie $\sqrt{2}R$ away from each other, meaning it looks like a random distribution, for which the direct application of the spherical LSH is efficient.

Since we divide our algorithm into two parts - cap carving and dense cluster elimination - we have to analyze each part separately.

First, we want to show that caps that we generate decrease in size. This translates to radius of the ball enclosing this points filled cap being bounded by the some fraction of original radius R. E.g. $\tilde{R} \leq (1-\Omega(\epsilon^2))R$. Consider the cap generated by $\delta B(o,R)B(x,(\sqrt{2}-\epsilon)R))$. WLOG we assume $o = 0$ and $x = (R, 0, \ldots, 0)$. The analysis then simplifies to essentially asserting that since we picked a subset of points that are $(\sqrt{(2)} - \epsilon)R$ away from the point of interest (anchor point for cap), and hence are covered by a circle of radius $(\sqrt{(2)} - \epsilon)R$ centered at that anchor point we can always inscribe a circle of a radius $\leq (1 - \Omega(\epsilon^2))R$ to cover these points.

Next we want to show that dense clusters also decrease in size in reasonable amount of time. The proof of convergence of this relies on separate convergences of each of its subfunctions. The main one of which is the function that converts the dense clusters to a sphere using the annuli technique (called ProcessBall in the original paper), which is recursive. This function itself relies on the function that deals with points on the sphere (ProcessSphere), which is also recursive and a function that transforms regular R^d space into (Process). Thus, if we can show that if all 3 converge, then we can conclude that algorithm converges as well.

The paper sets up following constants for the proofs that we will use:

- $\epsilon = 1 \log\log\log n$
- $\delta = exp(-\log\log\log n)C$, where C is set to the value s.t. errors introduced by rounding by $\delta areminimized$

- $\tau = exp(-log\frac{2}{3}m)$

The original paper makes the following argument:

1. Show that under relaxed assumption of that rounding of distances (done to map points to a new sphere) introduces no errors to ProcessBall and converges at a certain rate, the convergence is bounded by a certain quantity $(O_c(\frac{1}{\epsilon^6}))$

2. Show that this bound holds for a regular, non-relaxed case of ProcessBall, which only differs by a small multiplicative error

3. Show that ProcessSphere and Process converge at a rate $O(\log n)$

To prove the first item we divide ProcessBall into two categories: category A - λ^* shrinking and category B - λ^* non-shrinking. We analyze each separately, after which we sum the number of calls of each.

To analyze category A, we use the quantity $k = \frac{\Delta R2}{r_2^2}$ as our guiding tool. This quantity will help us determine the split between two types of calls that ProcessBall falls into. k controls the rate at which ratio between new and old threshold (termed η) ratio grows.

On a higher-level quantity λ^* represents a lower bound on the $\frac{\Delta R2}{r_2^2}$. We skip past certain derivations to assert that there are a total of $O_c(\frac{1}{\lambda^*})$ shrinking calls. Further derivations show that $O_c(\frac{1}{\lambda^*}) = O_c(\frac{1}{\epsilon^4})$ of such calls.

After we similarly derive the number of non-shrinking calls, we arrive at the $O_c(\frac{1}{\epsilon^6})$.

Thus, we sum the number to get $O_c(\frac{1}{\epsilon^6}) + O_c(\frac{1}{\epsilon^4}) \leq O_c(\frac{1}{\epsilon^6})$ total calls.

The next item (small multiplicative error) is assumed to be the case by the authors through setting parameter C.

The last item is proven through showing that at each call the number of points within the pseudorandom component slowly decreases (by a constant strictly smaller than one) and that gives us bound of $O(\log n)$.

5.1.1 Space and query time

The overall expected space the data structure occupies is n^1+o˙c(1) .

Proof.

1. Discretization step of ProcessBall (which creates many sphere instances) induces the fact that any pP_0 can participate in several branches of recursion, we need to figure out the number of these branches, and the cost of each.

2. The point p can find itself only in bounded number of branches - n^o˙c(1). This is derived using branching factor $O_c(\frac{1}{\delta})$ (derived using algebra from number of calls of inner loop of the, ProcessBall) and $O_c^{(O(1))}$ number of calls to ProcessBall (derived in proof for convergence of ProcessBall), in the following manner:

$$\text{of branches} = O_c(\frac{1}{\delta})^{O_c^{(O(1))}} = n^{o_c(1)}$$

3. Since we figured out the number of branches, it's time to look how much each branch call costs: n^o˙c(1). This is derived using the fact that number of calls per branch is a multiple of number of invoked recursive calls of ProcessBall and each calls their own instance of Process and ProcessSphere, which take up O($\log n)space.Thus:$

$$\text{of calls per branch} = O_c(\log n^{-O(1)}) = O_c(\frac{\log n}{\epsilon^{O(1)}}) = n^{o_c(1)}$$

4. We then figure out the space consumption of each point per partition by combining number of branches and cost of each branch: $n^{o_c(1)}n^{o_c(1)} = n^2 o_c(1) = n^{o_c(1)}$

5. Finally we need to figure out additive cost of base cases within each branch call for every point: n^o˙c(1) (taken as a fact in the paper)

6. Combining above items we get total space consumption of $n^{o_c(1)}(n^{o_c(1)} + n^{o_c(1)}) = 2n^2 o_c(1) \approx n^{o_c(1)}$

5.1.2 Query time

We need to expand query time to all of its essential elements:

Query time $= E[$ number of nodes in the recursive tree$]\cdot$(cost of a hashing point to a partition + cost of base case)

$$E[\text{number of nodes in the recursive tree}] = (\text{branching factor} \cdot$$
$$\text{number of calls to Process Ball})^{\text{number of calls in recursion stack}} = (\frac{1}{\delta} \cdot \frac{\log n}{\tau})^{(O_c(\epsilon_{O(1)})}$$
$$= n^{O_c(1)}$$

Then $Query\ \ time = n^{O_c(1)}(n^{O_c(1)} + n^{O_c(1)}) \approx n^{O_c(1)}$ (costs taken theorems 2.3, 2.5, 3.1;)

5.1.3 Preprocessing costs

That said, the preprocessing step looks nothing like polynomial time! In particular, the sheer number of ball partitioning/clustering we have to do jumps out to us: doesn't "pick a point and check if there are X number of points within a certain radius" involve at least X number of distance calculations, not to mention, wouldn't we need to check literally every single point in the worst case?

We can reduce the number of such computationally expensive operations by sampling the data points, since we only need clusters with $\geq \epsilon^2 \tau n = n^{1-o_c(1)}$ points. Then, if we sample $n^{o_c(1)}$ points and find dense clusters within this sample, the VC dimension provides us with a probabilistic upper bound on the test error – which in this case would be the probability that we found a dense cluster within the sample but it didn't translate to a dense cluster "big enough" in the overall dataset. This upper bound then assures us that the sample is accurate enough most of the time,

And finally, the author makes a passing remark to the difficulty of maintaining a dynamic data-dependent hashing algorithms for ANN. In case of regular LSH, because the hash functions are data-independent (i.e. fixed once chosen), we have the option of adding more points simply by hashing them as well – the new point doesn't affect the hashes of the existing points. Clearly, this is not the case for data-dependent hashing algorithms. Practically speaking, most OLTP workloads that involve geospatial queries (especially Near Neighbour queries) only need low dimensions (e.g. coordinates on Earth as a pair of longitude latitude). However, if we did need a dynamic data-dependent schemes with "reasonable" maintenance costs, we can't help but wonder whether we could exploit the fact that the iterative partitioning outlined in this paper looks like a tree, i.e. could we possibly "add" new data points to any particular node? While we do have a relatively fast way of figuring out where the new point should be (i.e. querying the tree), my concern is that no matter how well the tree is balanced (assuming that's even possible), there would be a significant write amplification problem – after all, the path to the new point needs to all be updated with the addition of one point, which in general would involve $\log x$, where x is the number of nodes in the "tree". This write amplification problem essentially gets back to the author's point about there being replicated points: "In fact, it will be necessary for some points to be replicated in a few children..." (pg. 4). While he does later show that the degree of replication is $n^{o(1)}$, which is a fairly trivial value in terms of storage, it starts to really impact the write amplification due to this factor. If there exists a data-dependent LSH (which, since it is iterative, would have somewhat of a tree structure) that does allow for “cheap” inserts of data points, then it would have to be relatively “flat”, i.e. it would have to be able to find the answer to a query q in fewer partitions. Perhaps here, the aforementioned concern of the seemingly arbitrary choice of a sphere as the state space and the suggestion to have different spaces as a hyperparameter might come into play, as they might have different characteristics that lead to same or slower query time but in fewer partitions.

6 References

[AINR14] Alexandr Andoni, Piotr Indyk, Huy L. Nguyen, and Ilya Razenshteyn. Beyond locality-sensitive hashing. In Proceedings of the 25th Annual ACM-SIAM Symposium on Discrete Algorithms (SODA ’2014), pages 1018–1028, 2014.

[DIIM04] Mayur Datar, Nicole Immorlica, Piotr Indyk, and Vahab S. Mirrokni. Locality-sensitive hashing scheme based on p-stable distributions. In Proceedings of the 20th ACM Symposium on Computational Geometry (SoCG ’2004), pages 253–262, 2004.

[NK17] Maximilian Nickel, Douwe Kiela. Poincaré Embeddings for Learning Hierarchical Representations. In Proceedings of Neural Information Processing Systems. (NIPS 2017).

[MNP07] Rajeev Motwani, Assaf Naor, and Rina Panigrahy. Lower bounds on locality sensitive hashing. SIAM Journal on Discrete Mathematics, 21(4):930–935, 2007.