Bit-Level Metric Compression

Jimmy Ye
UCSD

Abstract

We survey a selected set of theoretical and empirical results in algorithms for metric compression, ranging from classical results in dimensionality reduction to modern, state-of-the-art algorithms. The most recent, recommended algorithm, among those surveyed, is QuadSketch [7], which has provably near-optimal performance while demonstrating good empirical results. In addition, we propose a simple improvement to this algorithm.

1 Introduction

The representation of points in high-dimensional space are critical in the performance of modern algorithms in data science and machine learning. In particular, the space requirements for these representations often have a significant impact on performance due to the high cost of communication between different processing and memory units. Dimensionality reduction and other forms of approximation may also have some benefits to the robustness of downstream algorithms, as they can remove (or at least, uniformly randomize) the noisiness of the finite sampling of a distribution.

We therefore consider the problem of compressing metrics defined on finite sets of high-dimensional points, without any assumptions about the distribution of the points. Equivalently, we want to construct a *sketch*, a data structure smaller than the original data, with which we can approximate the metric on the original set. For simplicity, we consider only Euclidean distance, though many of the results presented can be generalized to other, or even arbitrary, metrics.

2 Dimensionality reduction as compression

A well-known classical result in dimensionality reduction is the Johnson-Lindenstrauss (JL) lemma. [1][2] It is of interest because it demonstrates that metric compression can rely only on the size of the point set and the desired error of approximation, with no dependency on the dimensionality of the data. Intuitively, the JL lemma defines a projection which preserves the structure of the data set (as given by the metric - it does not address whether a chosen metric captures the properties desired), ignoring the “details” of the ambient space in which the points are embedded.

Many variants of the JL lemma exist. We formally state one such variant [2]; the minor differences between the variants are not relevant for our purposes:

Theorem 1 *Let $\epsilon \in (0, 1/2)$, and let $X = \{x_1, x_2, ..., x_n\}$ in $\mathbb{R}^d$ be a set of high-dimensional points. For some positive integer $k = O(\log(n)/\epsilon^2)$, there exists a matrix $A \in \mathbb{R}^{k \times d}$ such that for any $x, y \in X$, we have*

$$(1-\epsilon)\|x-y\|_2^2 \leq \|Ax-Ay\|_2^2 \leq (1+\epsilon)\|x-y\|_2^2$$

We note that, without loss of generality, we may remove (i.e. replace with 1) the term $(1-\epsilon)$ in the left inequality, by a simple scaling argument. This general form of inequality is referred to as a *distortion* of $1+\epsilon$, and since the two forms are equivalent except for a constant factor, we refer to both as $1+\epsilon$ distortion without distinction.

Preprint. Work in progress.

It has also been proven that the JL lemma is tight, i.e. there is some "hard" set of points for which achieving $1+\epsilon$ distortion requires $O(\log(n)/\epsilon^2)$ dimensions. [6] This will later be useful for proving certain lower bounds on space required.

While the JL lemma provides a strong result in dimensionality reduction, it does not give us a bound on the exact space requirements. The algorithm presented in [3] provides us with a concrete, practical algorithm, with the added benefit of allowing a simple analysis of the space complexity. We summarize the relevant parts of this result:

Theorem 2 *Let $\beta > 0$. We set the parameter $k = O(\beta \log(n)/\epsilon^2)$, with all other parameters defined as in the JL lemma.*

Define an embedding using a random matrix $A \in \{-1, 0, 1\}^{k \times d}$ with entries sampled from the distribution given by probabilities $P(-1) = 1/6, P(0) = 2/3, P(1) = 1/6$.

After scaling A by constants, this gives an embedding of X with $1+\epsilon$ distortion with probability $1 - n^{-\beta}$.

We see immediately that this algorithm is simple to implement, as in [3] which proposes using a database aggregation query, and it provides strong concrete bounds in line with the JL lemma. It also has the benefit of requiring very few random bits.

Of particular interest for our purposes, a simple space analysis immediately follows. Given a precision parameter Φ (the ratio of maximum to minimum distance, also referred to as the *spread*), the precision of scalars in the original dataset is $\log \Phi$ bits. Since we sum over all $O(n)$ points for each projected coordinate, our expected and worst-case space is $O(\log(n\Phi))$ bits per coordinate (BPC), giving us $O(\log n \log(n\Phi)/\epsilon^2)$ bits per point (BPP). This gives us our classical baseline for metric compression.

3 Near-optimal compression

We present a more recent set of results from [4]. They show that the JL baseline is suboptimal, and indeed, rather far from optimal. They provide an algorithm which significantly improves on the baseline, using a variant of hierarchically well-separated trees [5], and prove that this algorithm is asymptotically near-optimal. (Note that this algorithm compresses the given metric without dimensionality reduction; the bounds given are therefore based on first applying a JL embedding like our baseline.) As the details are complicated, we only provide a simplified, high-level overview, similar to one given in [4].

3.1 Algorithm overview

In essence, we can construct a tree T of hierarchical clusters. Each level of the tree corresponds to a partition based on a clustering. At level l, the clustering is given by connecting all pairs of points where the distance between them is at most 2^l. From each cluster, a "center" point is chosen.

This tree can be compressed by contracting long non-branching paths. We can view this as replacing a cluster with its center when estimating distance between any point internal to the cluster and a point external to the cluster. They show that the criteria for this contraction implies that the points are clustered, or equivalently, that the cluster is well separated from the rest of the points, such that this approximation is sufficiently tight.

The next step is to represent points as displacements from nearby points. A cluster at level l is formed from edges between a number k of clusters at level $l-1$. In this graph a rooted spanning tree is constructed, and every non-root cluster is represented as the displacement from its parent, referred to as the "ingress" of the child. This distance is measured by the distance from the child's center to the closest point in the parent.

Finally, these displacements are rounded, with a precision dependent on diameter and level. The approximation given by adding the rounded displacement to the "ingress" is a "surrogate", and distances are approximated by the distances between corresponding surrogates.

This description is a simplification in many ways, including the choices of ingress and surrogates.

3.2 Algorithm results

We present the results of [4] regarding this algorithm, and its near-optimality, though for brevity we omit most of the proof details.

Theorem 3 *The algorithm described has worst-case space complexity (in BPP) of* $O(\log(1/\epsilon)\log(n)/\epsilon^2 + \log\log\Phi)$.

This replaces the $\log(n\Phi)$ term of the baseline by $\log(1/\epsilon)$. In practice this tradeoff is often of immediate benefit, but perhaps just as importantly, for many applications, n may grow arbitrarily based on external factors, whereas the tolerance ϵ is fixed. The additive term $\log\log\Phi$ is, by comparison, almost negligible.

In practice we must also consider the time complexity for both construction and querying, for which the following results hold:

Theorem 4 *The algorithm described for constructing the sketch has time complexity* $O(n^2\log\Phi + nd^{1+1/p}\log(1/\epsilon)/\epsilon)$*, for* d*-dimensional data under the* L_p *metric.*

For Euclidean space with a JL embedding, we obtain $O(n^2\log\Phi + n(\log^{3/2} n)\log(1/\epsilon)/\epsilon^4)$.

Additionally, in Euclidean space, we may reduce this time complexity in exchange for increasing the space. Specifically, the n^2 *can be reduced to* $n^{1+\alpha}$*, in exchange for increasing the space used by a multiplicative factor of* $1/\alpha$.

The space-time tradeoff is obtained by using locality-sensitive hashing (LSH). By computing the clusters in the tree approximately using LSH (increasing the distances allowed for merging from 2^l to $\alpha^{-1/2}2^l$), we replace the n^2 time with $n^{1+\alpha}$, but to compensate for the approximation, ϵ is scaled by $\alpha^{1/2}$, giving a multiplicative increase in space of $1/\alpha$.

Theorem 5 *The querying algorithm, which involves traversing the constructed tree, among other steps, has a time complexity of* $O(\log\Phi + nd^{1+1/p}\log(1/\epsilon)/\epsilon)$*, for* d*-dimensional data under the* L_p *metric. For Euclidean space with a JL embedding, this is* $\tilde{O}(n/\epsilon^4)$.

This time complexity may be reduced to $O(\log\Phi + d\log\left(\Phi d^{1/p}/\epsilon\right))$ *in exchange for increasing the space by a factor of* $\log d$.

For Euclidean space with a JL embedding, and the simplifying assumption that $\Phi = \mathrm{poly}(n)$*, this increases the size by* $O(\log(1/\epsilon) + \log\log n)$ *and achieves* $\tilde{O}(\log^2 n/\epsilon^2)$ *query time.*

Note that $\tilde{O}$ notation is used here, to ignore polylogarithmic factors.

As compared to the space-time tradeoff for construction, the space-time tradeoff for querying is essential. A query time which is linear in n and quartic in $1/\epsilon$ is almost always infeasible, especially for the sorts of applications where we would typically want to use metric compression methods. The increase in size is reasonably small, in exchange for queries which are so much faster as to be considered a cheap operation.

3.3 Lower bound

We present the result of [4] giving a lower bound on metric sketch size:

Theorem 6 *For any* $\gamma > 0$*, if* $\epsilon \geq 1/n^{0.5-\gamma}$*, then the sketch size produced by an algorithm which works for arbitrary Euclidean distances (regardless of the original dimension) is lower bounded by* $\Omega(\gamma\log(n)/\epsilon^2 + \log\log\Phi)$ *BPP.*

In particular, this result shows that, up to fixing a reasonable γ, the given algorithm is only a factor $\log(1/\epsilon)$ away from optimal.

3.3.1 Proof

They show this result by a direct construction of two possible point sets, one for each of the terms, and then combine the two to achieve their sum.

For the first, an arbitrary sequence of n vectors is chosen. These vectors are generated by taking k-sparse vectors chosen from $\{0,1\}^n$, and normalizing them (i.e. scaling by $1/\sqrt{k}$) but we consider them as points in $\mathbb{R}^n$ with the usual Euclidean metric. In addition to these vectors, we can take the standard basis vectors. Then with a $1 \pm (\epsilon/2)$ distortion metric sketch, it can be shown that we can recover the original sequence of n vectors exactly. Since there are $\binom{n}{k}^n$ possible choices for this sequence, the metric sketch must be at least $\log\left(\binom{n}{k}^n\right)$ bits, which is further lower bounded by $\Omega(\gamma n \log(n)/\epsilon^2)$.

For the second term, a similar approach is followed by embedding $X = \{1, ..., n\}$ in $\mathbb{R}$, where the embedding f is fixed for $f(1) = 0$ and otherwise chosen arbitrarily from $\{2, ..., 2^{log\Phi - 1}, \Phi\}$. This family of metrics has $(\log\Phi)^{n-1}$ members, and is recovered exactly by a metric sketch of distortion less than 2. Thus it takes $\Omega(n \log\log\Phi)$ bits to sketch these metrics.

Taken together, we can form a family of metrics of $2n$ points in $n+1$ dimensions and spread Φ. Scaling by $1/2$, taking large enough n, and the lower bound on ϵ dependent on γ, we can embed this family of metrics exactly in $O(log(n)/\epsilon^2)$ dimensions. By recovering these metrics exactly, we have shown the lower bound of $\Omega(\gamma n \log(n)/\epsilon^2 + n \log\log\Phi)$ total bits, i.e. $\Omega(\gamma \log(n)/\epsilon^2 + \log\log\Phi)$ BPP.

3.4 Remarks

Demonstrating the lower bound for metric compression is a strong theoretical result, as is the near-optimal algorithm provided. However, the algorithm presented is quite complex, and has not been implemented as of December 2017. [7]

It also requires significant additional changes to achieve reasonable construction and query times, and even after doing so, the construction time scales super-linearly with $O(n^{1+\alpha})$ with a space increase of $1/\alpha$. For even reasonably large n, this is a rather severe penalty even for the optimal choice of α (asymptotically, $\log^{-1} n$).

4 Practical metric compression

In practice, there are many other approaches to metric compression. However, the result of [7] is of particular interest, because the algorithm presented is not only provably near-optimal, building on the work of [4], but also much simpler, and with good empirical performance. We first present their base algorithm, QuadSketch.

4.1 QuadSketch

For input data X and parameters L and Λ (which determine ϵ as well as the time and space complexities), the QuadSketch construction algorithm is as follows.

1. Let x_i be an arbitrary point in X. Let Δ be the maximum distance from x_i to any other point in X. Then we shift each coordinate j by a corresponding $\sigma_j \in [-\Delta, \Delta]$ chosen uniformly randomly. It is noted that while this is required for theoretical guarantees, it is not typically necessary in practice.

2. We construct a "quadtree", or rather, technically, a 2^d-ary tree, simply by repeatedly partitioning a hypercube H containing X into 2^d parts, in the straightforward generalization of partitioning a square into quadrants for a quadtree in $\mathbb{R}^2$. Note that we only actually construct nodes of the tree which contain at least one x_i, and that each edge is labeled by d bits in the obvious way to identify one of at most 2^d partitioned parts. Denote this tree T^*.

3. Let $u_0, u_1, ..., u_k$ be any path in T^* where u_0 and u_k have degree (number of children) not equal to 1, all other nodes have degree 1, and $k > \Lambda + 1$. We define the tree T by contracting the tail of this path such that u_k becomes the child of u_Λ. Edges in T are either contracted ("long") or uncontracted ("short").

4. For each $x_i \in X$, store the corresponding leaf v_i. Encode T using the Eulerian Tour Technique, i.e. a DFS where each step is encoded as a single bit (downward/upward). Edge information is encoded with downward steps, with 1 bit for marking an edge as short or long, followed by either an exact relative location for a short edge, or the length of the edge for a long edge.

To recover a point approximately from this tree encoding, traverse the tree from the root to the leaf, collecting either the bits of short edge labels, or appending as many zeroes as the edge length of a long edge. This gives the binary representation of the approximated point.

4.2 Results

For QuadSketch as described above,

Theorem 7 *Given $\epsilon, \delta > 0$, let $\Lambda = O(\log(d \log \Phi / \epsilon\delta))$, $L = \log \Phi + \Lambda$. QuadSketch runs in $\tilde{O}(ndL)$, produces sketch with $O(d\Lambda + \log n)$ BPP, such that distances are preserved in the sense that, for any x_i, we have*

$$P(\forall_{x_j} (1-\epsilon)\|x_i - x_j\| \leq \|\tilde{x}_i - \tilde{x}_j\| \leq (1+\epsilon)\|x_i - x_j\|) \geq 1 - \delta$$

That is, all distances to any x_i are preserved with $1+\epsilon$ distortion with probability $1-\delta$, for each x_i independently.

For Euclidean space with JL embedding, and $\Phi = \text{poly}(n)$, we obtain $O(\log \log n + \log(1/\epsilon))$ BPC and runtime $\tilde{O}(n/\epsilon^2)$.

Note that the construction time complexity no longer has any factors of n^c with $c > 1$, i.e. nearly linear in n, which is a drastic improvement over the other, much more complex near-optimal algorithm based on hierarchically well-separated trees.

However, we can see that the distortion guarantees here are slightly weaker, each point has its own independent distortion. In this formulation, δ would have to be extremely small for reasonably large n to get all points to simultaneously achieve $1+\epsilon$ distortion with high probability. Additionally, the BPC is slightly worsened, by an additive term of $\log \log n$.

Thus, the following result from [7] (stated slightly more explicitly) was proven, to obtain the same guarantees across the entire metric sketch as before:

Theorem 8 *Given $\epsilon > 0$, let Λ and L be defined as in Theorem 7. There is an algorithm which performs $O(\log^2 n)$ iterations of QuadSketch, achieves the time and space complexity guarantees of Theorem 7, and with high probability, produces a sketch with distortion at most $1+\epsilon$.*

There is a notable disadvantage to this algorithm: since it produces multiple randomized QuadSketch sketches and approximates any given distance by selecting one of these sketches at a time, it does not give us an embedding from which an approximation for X as a whole can be derived. It only allows us to query for distances $\|x_i - x_j\|$.

4.3 Proof sketch

The outlines of the proofs given in [7] of Theorem 7 and Theorem 8 are as follows:

We can define a point x_i as being (ϵ, Λ)-padded if, for every level l of the "quadtree" T, the node / grid cell containing x_i also contains a ball centered at x_i of a certain radius $\rho(\epsilon, \Lambda, l)$ (omitted here for brevity).

Then we can show that (1) with the random shift performed as the first step of QuadSketch, every point x_i is (ϵ, Λ)-padded in T with probability $1-\delta$ and (2) any point which is (ϵ, Λ)-padded will have distances preserved with $1+\epsilon$ distortion.

The proof of (1) is relatively simple, and follows from taking a union bound over the random shift in each coordinate, which makes it relatively unlikely that the point will be close to a cell boundary.

The proof of Theorem 8, the recursive variant of QuadSketch, simply relies on repeating QuadSketch on a subset of X which is not yet padded, until all points in X are padded. Then for any pair of points, the point which is padded earlier (or in the same iteration) will always be in a tree with the other point, giving a distance with sufficiently low distortion.

By Markov's inequality, setting δ sufficiently low will ensure a relatively high probability of a large proportion of the points being padded. For convenience, we can set $\delta = 0.25$ to get a probability of 0.5 that half the points are padded, giving a joint probability of $O(1/n)$ of this happening in each of $O(\log n)$ iterations (which is sufficient to make all the points padded). By a further $O(\log n)$ independent repetitions of this, we can increase the probability of padding all the points from $O(1/n)$ to a high constant probability. Thus $O(\log^2 n)$ repetitions of QuadSketch will give us the desired result - all points being padded in some sketch, with high probability.

Note the time complexity in Theorem 7 ignores polylogarithmic factors, and each iteration at least halves the number of points. Thus the complexity guarantees of Theorem 7 are maintained for this recursive application of QuadSketch.

4.4 Experiments

Multiple experiments were performed in [7] which evaluated QuadSketch in the form of the "block" variant of the basic non-recursive algorithm, i.e. partitioning the dimensions into m blocks on which the algorithm is performed independently.

QuadSketch was compared to another compression scheme, Product Quantization [8], as well as a baseline of simple uniform quantization, where each coordinate, having some interval from a minimum value to a maximum value, is partitioned by the placement of k equally spaced values. 3 common benchmark datasets including MNIST [9] were used, as well as a synthetic dataset consisting of random points along the diagonal line $(x, x, ..., x)$.

The results were reported (scatter plotted) for many different parameter combinations of Λ, L, m. For the most part, QuadSketch was competitive with or markedly superior to Product Quantization.

The source code for their implementation was provided at [10].

4.5 Remarks

This set of results from [7] is quite strong: the algorithm is practically simple enough to implement from a description that fits on a page or less, the provable bounds on its performance are near-optimal, and there are empirical results which demonstrate reasonable improvements compared to prior work.

However, one particular omission is further detail on the algorithm for decompression / querying, especially as compared to [4], which goes into detail on how to improve from linear runtime to polylogarithmic. A straightforward reading of [7] seems to imply that on average, a query has linear runtime w.r.t. n, as $O(n)$ tree nodes (i.e. on average $n/2$) must be traversed to reach a given leaf. Of course, in batch, many nodes may be decompressed in a single traversal, but even in this case, we may end up with most of the traversal wasted. And in many cases, singular or very small numbers of queries may be desired.

5 Improving query times

5.1 Preliminaries

For this section, we make the simplifying assumptions of only working with Euclidean space, with a JL embedding (i.e. $d = O(\log(n)/\epsilon^2)$), and $\Phi = \text{poly}(n)$ (i.e. $\log \Phi = O(\log n)$). The analyses can be generalized to an arbitrary metric, dimension, and precision, but we do not do so here.

In addition, recall that the pruning parameter is set to $\Lambda = O(\log(d \log \Phi/(\epsilon\delta)))$, and the quadtree has $L = \log \Phi + \Lambda$ levels. [7]

Based on the simplifying assumptions, $\Lambda = O(\log(\log n/(\epsilon\delta)))$. We also further assume that $1/(\epsilon\delta)$ is sufficiently small so that $\Lambda = O(\log n)$, and thus $L = O(\log n)$.

5.2 Algorithm

Improving the query runtime can be done through a straightforward adjustment to the tree encoding scheme. For every node p in the tree, denote its children u_i, where i is determined by the ordering of the nodes in the encoding (the same order as given by [7]).

Then let $N_{u_i} = \sum_{j=1}^{i} N_j$ where N_j is the number of nodes at the subtree rooted at u_j, and let V_{u_i} be the index (corresponding to order in the entire tree, as in [7]) of the last leaf in the subtree rooted at u.

For every node p and its children u_i, we encode (1) the degree of p, followed by (2) N_{u_i} and V_{u_i} for each u_i.

Note that by encoding information for a node p, we refer to directly after the incoming edge to that node in the Eulerian Tour encoding given in [7]. For the root node of the tree, we simply store this information at the beginning of the tree encoding.

During the traversal, when querying for a leaf v_i and visiting a node p, we can perform binary search over the children via the degree of p and V_{u_i}, to find which node u_i that v_i is under. Then we can skip directly to u_i using the number of nodes N_{u_i} and the number of bits per node/edge. (We assume here that long edges are padded to d bits, to get a constant number of bits for all edges.)

5.3 Correctness and space

Since the v_i we are querying for always corresponds to some x_i, and is therefore always in the tree, there is exactly one path to v_i from the root. Thus for any internal node p on this path, exactly one of its children u_i is on that path. Since we store V_{u_i} for all children u_i, and the encoding is in DFS order, we can identify the correct u_i as the earliest u_i such that $V_{u_i} \geq v_i$. A binary search allows us to do exactly that, since the sequence of V_{u_i} is sorted by construction (as leaf indices v_i increase with u_i in DFS order).

Then, by padding long edges from $\log n$ bits to $d = O(\log n/\epsilon^2)$ bits, we obtain a constant number of bits for every edge: 2 bits for going up/down the edge in the Eulerian Tour, $O(\log n)$ bits for the N_{u_i} and V_{u_i} corresponding to the child node of the edge which are stored in the parent node of the edge, and $d + 1$ bits for the edge label (short/long, coordinates/length). Since the number of nodes in a subtree is exactly equal to the number of edges to be skipped, i.e. all the internal edges plus the incoming edge to the root of the subtree, N_{u_i} is sufficient to calculate the number of bits to skip ahead.

By the same analysis as [7], since the added information makes the long edges the same length in bits as the short edges, we increase the space required by only a constant factor, i.e. we maintain $O(d\Lambda + \log n)$ BPP.

Note that we can obtain a small constant factor improvement by also storing the number of long edges in a subtree, rather than padding the long edges.

5.4 Time

We consider the following operations as constant time: (1) an edge traversal, (2) comparing two $O(\log n)$ bit integers. These are both upper bounded by $O(d)$ bit operations for $\log n = O(d)$, but are constant across any scheme using a tree of this structure, since we always need to construct a $(d \log \Phi)$-bit approximation for a point by traversing $O(L)$ edges from the root.

In [7], the decompression of an individual point takes up to $O(n\Lambda)$ operations, as all edges must be traversed to reach the last leaf. Likewise, decompressing all of the points also takes $O(n\Lambda)$ edge traversals.

With this improvement to querying, we can decompress a single point in $O(\log^2 n)$ total operations. Every branching node has at most n children, since by construction every child leads to a leaf which corresponds to a point from the original dataset. We can binary search over n children in $O(\log n)$ comparisons, and the tree has $L = O(\log n)$ levels, so there are $O(\log^2 n)$ comparisons, which dominates the $O(\log n)$ edge traversals. Batch queries of v points will take at most $O(v \log^2 n)$ operations.

We can show a tighter bound using the arithmetic-geometric mean (AM-GM) inequality. Let $D_1, D_2, ..., D_k$ be the degrees of each branching node on the worst-case path to a leaf, where by construction, $k \leq L$. Then the required number of comparisons is $\sum_{i=1}^{k} \log_2 D_i = \log_2(\Pi_{i=1}^{k} D_i)$. We can upper bound this using the AM-GM inequality:

$$\frac{1}{k}\sum_{i=1}^{k} D_i \geq \sqrt[k]{\Pi_{i=1}^{k} D_i} \tag{1}$$

$$\log\left(\frac{1}{k}\sum_{i=1}^{k} D_i\right) \geq \frac{1}{k}\log\bigl(\Pi_{i=1}^{k} D_i\bigr) \tag{2}$$

$$k\log\left(\frac{1}{k}\sum_{i=1}^{k} D_i\right) \geq \sum_{i=k}^{k} \log D_i \tag{3}$$

Equality is achievable precisely when all D_i are equal, so let $D_1 = D_2 = ... = D_k = D$. In particular, the LHS simplifies to $k \log D$.

Note that $\sum D_i \leq n + k$, since there are k branching nodes in the path, and every child at each branch which is not on the path will lead to at least one distinct leaf. This upper bound is achieved when each leaf (other than the one at the end of the path) corresponds to a unique child of a branching node.

Thus we obtain $D \leq \frac{n+k}{k}$, and therefore the LHS of inequality (3) is maximally $L\log\bigl(\frac{n+L}{L}\bigr)$. Asymptotically, $L = \Theta(\log n)$ in the worst case, so the worst-case decompression time is $\Theta(\log n \log \frac{n}{\log n})$.

However, in most cases, we will find that decompression of a single point actually takes $O(\log n)$ operations. Consider datasets which are well-clustered, i.e. the degree of every node is upper bounded by a small constant independent of n. In this case, binary search across the children of each node is $O(1)$, giving an overall decompression time of $O(\log n)$.

Then consider the opposite, datasets which are well-separated in the sense that branching nodes have degree $\Omega(n)$. In this case, the number of branching nodes which is encountered on any path to a leaf must be $O(1)$. Thus there are $O(\log n)$ comparisons required, so decompression takes $O(\log n)$ operations under our assumptions.

Thus in both cases of well-clustered and well-separated data, we obtain $O(\log n)$ operations per point for decompression, which is significantly better than the $O(\log^2 n)$ operations of the pathological worst case. We can also see that this applies, via recursion on subtrees, to any tree where every branching node is either well-clustered or well-separated.

5.5 Conclusion

We have presented a modified algorithm which generally improves upon [7] for query time. For most reasonable datasets, i.e. those satisfying our original simplifying assumptions, we achieve an improvement from querying time linear in the size of the dataset to polylogarithmic, with only a constant factor increase in space. We have also shown that it is realistically possible to impose additional requirements, such as the data being well-clustered or well-separated, to obtain an improvement from $\log^2 n$ to $\log n$.

6 References

[1] W.B. Johnson, J. Lindenstrauss, Extensions of Lipschitz mappings into a Hilbert space, Conference in modern analysis and probability, New Haven, CI, 1982, Amer. Math. Soc., Providence, RI, 1984, pp. 189–206.

[2] CSE 291 Lecture 5

[3] D. Achlioptas. Database-friendly random projections: Johnson-Lindenstrauss with binary coins. Journal of Computer and Systems Science, 66(4):671–687, 2003. Special issue of invited papers from PODS'01.

[4] P. Indyk and T. Wagner. Near-optimal (euclidean) metric compression. In Proceedings of the Twenty-Eighth Annual ACM-SIAM Symposium on Discrete Algorithms, pages 710–723. SIAM, 2017.

[5] Y. Bartal. Probabilistic approximation of metric spaces and its algorithmic application. Proc. 37th FOCS, pages 184–193, 1996.

[6] Kasper Green Larsen and Jelani Nelson. Optimality of the Johnson-Lindenstrauss lemma. In Proceedings of the 58th Annual IEEE Symposium on Foundations of Computer Science (FOCS), 2017.

[7] Piotr Indyk, Ilya Razenshteyn, and Tal Wagner, Practical data-dependent metric compression with provable guarantees, Advances in Neural Information Processing Systems, 2017, pp. 2614–2623

[8] H. Jegou, M. Douze, and C. Schmid. Product quantization for nearest neighbor search. IEEE transactions on pattern analysis and machine intelligence, 33(1):117–128, 2011.

[9] Y. LeCun and C. Cortes. The mnist database of handwritten digits, 1998.

[10] `https://github.com/talwagner/quadsketch`