# Online Nearest Neighbor Search in Hamming Space

Sepehr Eghbali, Hassan Ashtiani, Ladan Tahvildari
University of Waterloo, Ontario, Canada
Email: {s2eghbal, mhzokaei, ladan.tahvildari}@uwaterloo.ca

***Abstract*—We address the $K$ Nearest Neighbors ($K$NN) problem in large binary datasets which is of major importance in several applied areas. The goal is to find the $K$ nearest items in a dataset to a query point where both the query and the items lie in a Hamming space. We address this problem in its online setting, that is, data items are inserted sequentially into the dataset. We propose a data structure that partitions the feature space by exploiting the Hamming weights of the binary codes and their substrings. The proposed data accommodates efficient similarity search and fast insertion of new items. Empirical evaluations on large-scale datasets used in binary hashing techniques show significant speedup over the best known solutions.**

## I. Introduction

Recent decades have witnessed a growing surge of research on representing large datasets with binary features. A typical example is the representation of text documents as *term vectors*, where absence or presence of words (or shingles) are captured with binary variables [1], [2]. Also, in machine vision, binary feature extractors have been used to describe images [3], [4].

Perhaps the most notable application of binary codes is the *binary hashing* [5] which has garnered much interest in the last decade. Binary hashing techniques aim at encoding high dimensional vectors with *compact* similarity preserving binary codes (vectors). Such codes are mainly used for the task of nearest neighbors search as they facilitate storage and benefit distance computation. There exists a large body of work on generating compact similarity-preserving binary codes for the purpose of fast retrieval [6], [7], [8], [9], [10]. The performance of binary hashing techniques is measured in terms of the length of the generated code and fidelity to the similarity measure. However, the problem of efficiently searching in large-scale binary datasets is still a bottleneck for fast retrieval.

The task of finding the closest neighbor to a given query point in the Hamming space, known as the *Hamming nearest neighbor* (HNN) problem, arises as a core procedure in many applications of binary datasets such as machine learning, information retrieval and data mining. This problem admits a straightforward solution–linearly scanning the entire set of items–but for today's large-scale datasets, the linear scan time can take more than several minutes [11].

Modern real-life datasets are not only large in the number of data points, but also are open-ended and dynamic: new items appear and are added to the dataset over time. For example, a search engine often has numerous new web pages containing images and textual data, that are continuously arriving at the data center everyday. In online HNN therefore, the nearest neighbor queries must be answered based on the total data that has been gathered so far. This leads to a natural question: *can we perform better than linear scan (search) for the task of large-scale HNN in online settings?*

This question has not been answered adequately in the literature. Some recent researches have addressed the problem of learning compact binary codes in online settings [12], [13]. They have shown that it is possible to gradually update the hash function (that maps real data to binary codes), as the data items becomes available, such that the function can better preserve the similarities. However, the problem of efficiently searching among the so-far collected binary codes seems to have remained unsettled. In practice, researches often resort to linear scan to find nearest neighbors in online applications. In this study, we revisit the exact nearest neighbor search in the Hamming metric for compact binary codes. We propose a data structure for solving the online HNN problem with efficient search and insertion time. The main contributions of the paper are as follows:

- We propose the *Hamming Weight Tree* (HWT), a data structure for partitioning the feature space based on the Hamming weights (i.e. $\ell_1$ norm) of the binary codes.
- We develop two algorithms to search for the nearest neighbor and to insert new items to the tree.
- We empirically evaluate the performance of the proposed data structure on large binary datasets. The results demonstrate several orders of magnitude speed up for HWT in comparison with linear scan and better average performance compared with the state of the art for batch HNN.

## II. Background Review

The nearest neighbor search problem in the Hamming space, originally presented by Minsky and Papert [14], has been extensively studied in the literature both because of its theoretical importance and because of its numerous applications that abound in image retrieval [6], [15], duplicate detection [16] and matching local features [17]. From the theoretical standpoint, like many other proximity problems, nearest neighbor search (even in the Hamming space) suffers from the *curse of dimensionality* phenomenon: as the number of dimensions increases, all algorithms would be inferior to linear scan [18]. Unfortunately, to this day, there is no algorithm with polynomial pre-processing and storage costs which guarantees sublinear query time [19]. Some studies have focused on *approximate* solutions that trade accuracy for scalability [20]. This line of work managed to break the linear query time bottleneck and had enormous impact. However, this paper is primarily concerned with the exact search problem.

Despite the discouraging theoretical results for finding the exact solution of HNN, vast empirical evidence strongly suggests that it is possible to perform better than linear scan in many cases and achieve acceptable search time on standard benchmarks [21], [22], [23]. From this perspective, several studies have provided practical solutions for the HNN problem in the static setting. Space-partitioning algorithms such as $kd$-tree and Voronoi diagram are among the best known nearest neighbor algorithms for low dimensional spaces (up to 20 or 30) [24]. However, the query time of such techniques degrade exponentially with the number of dimensions. Moreover, most of space-partitioning techniques for the task of nearest neighbor search do not support dynamic datasets [21].

Since binary codes lie in a discrete space, some researches have mentioned the use of hash table to reduce the search cost. The idea is to populate a hash table with binary codes of dataset and then probe the buckets within some ball around the query to find the nearest neighbors [25]. This approach is also interesting in online settings as the amortized cost of inserting a new item into hash tables is constant. However, for long codes, vast majority of buckets are empty and in turn the search algorithm must increase the radius until the ball around query hits a point. Unfortunately, oftentimes, the required radius of search is large enough to make the number of probed buckets exceed the total number of points in the dataset. This issue turns linear scan into a faster alternative.

Multi-index hashing (MIH) [23] is a rigorous approach for handling this issue. The key idea of MIH is that two similar binary strings must also have similar substrings. Therefore, rather than populating a single huge hash table, MIH builds multiple smaller hash tables on the substrings of binary codes. To find the nearest neighbors, the query is similarly partitioned into several substrings and search is performed in each hash table independently. Empirical experiments show that MIH can provide dramatic speed-up over the linear scan baseline. Recently, Ong and Bober [11] proposed an algorithm for tuning the length of substrings assigned to each hash table.

Despite its success, the performance of MIH heavily depends on knowing the number of dataset items beforehand. In [23] and [26], the empirical analysis shows that the best performance of MIH is often achieved when for every $\log_2 n$ bits (where $n$ is the number of items), one hash table is constructed. Just as importantly, the same analysis indicates that setting the number of hash tables to a wrong number can incur extra work, even significantly more than what is required for the linear scan. Consequently, MIH is mostly applicable for batch data in which the number of items remains constant and known.

## III. Hamming Weight Tree

We address two closely related problems. Given a dataset of binary codes $\mathcal{B} = \{\mathbf{b}^i \in \{0,1\}^p\}_{i=1}^n$, and a binary query vector $\mathbf{q} = \{0,1\}^p$, the first problem is the *$r$-neighbor* problem or *range query*, whose goal is to report all codes in $\mathcal{B}$ that are within a given distance $r$ from $\mathbf{q}$. The second problem, $K$ nearest neighbor, aims at finding the $K$ codes in $\mathcal{B}$ that are closest to $\mathbf{q}$ in terms of the Hamming distance. We address both problems in their online settings; that is, the items in $\mathcal{B}$ become available sequentially and the size of dataset is unknown.

### A. Depth One Tree

We first propose a data structure for solving the $r$-neighbor problem and then apply the data structure to solve the $K$NN problem. The key idea of this paper rests on the following statement: when two binary codes $\mathbf{h}$ and $\mathbf{g}$ differ by at most $r$ bits then the difference between their Hamming weights is at most $r$ where the Hamming weight of a binary code is the number non-zero entires in the code. This leads to our first proposition:

*Proposition 1:* If $\|\mathbf{h} - \mathbf{g}\|_H = r$, then we have:

$$r - |\|\mathbf{h}\|_H - \|\mathbf{g}\|_H| \in \{0, 2, \ldots, r-2, r\}. \tag{1}$$

where $\|.\|_H$ denotes the Hamming weight (Proof in Appendix A).

It is easy to see that based on Proposition 1, for two binary codes with Hamming distance of at most $r$ ($\|\mathbf{h} - \mathbf{g}\|_H \leq r$), the difference of Hamming weights is also at most $r$. In other words, the difference of Hamming weights, or the *Hamming weight distance*, is a lower bound for the Hamming distance. Computing the Hamming weight is an extremely fast operation as many of the modern CPUs provide *popcnt* (population count) instruction which implements the Hamming weight function at the hardware level.

The significance of (1) stems from the fact that to solve the $r$-neighbor search problem for the given query $\mathbf{q}$, one needs to retrieve binary codes with Hamming weights in the set $\{\|\mathbf{q}\|_H - r, \ldots, \|\mathbf{q}\|_H + r\}$ and ignore the rest of the points. Unfortunately, the retrieved codes are not restricted to the Hamming radius of interest around the query. Hence, not all items in the target sets are $r$-neighbors of the query, so we need to cull any candidate that is not a true $r$-neighbor.

For example, to answer a 2-neighbor problem for the query code $\mathbf{q}$ on a dataset of 128-bit binary codes, we can create a tree, which we call the *Hamming Weight Tree*, with 129 leaves (one for each possible Hamming weight), and assign the codes of dataset to their corresponding leaf node based on their Hamming weights (see Figure 1). Assuming that we have $\|\mathbf{q}\|_H = 64$, to answer the 2-neighbor query, the algorithm linearly searches among the codes belonging to nodes 62,63,64,65,66 and ignores the other 124 nodes. More generally, to solve 2-neighbor query for any query point, the algorithm needs to check at most five leaf nodes.

To create the HWT, the pre-processing step of our algorithm partitions the binary codes of $\mathcal{B}$ into $p+1$ sets, each corresponding to one of the possible Hamming weights. Then, during the query phase, the algorithm retrieves the points in the nodes whose Hamming weight difference from $\mathbf{q}$ is at most $r$. Interestingly, inserting new items to the HWT is easy as we only need to compute the Hamming weight of the new code and add it to the corresponding leaf.

An ideal scenario for solving the nearest neighbor problem using the HWT occurs when the algorithms only needs to

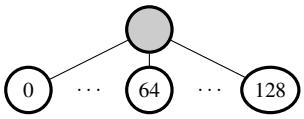


Fig. 1. A Hamming weight tree with depth one

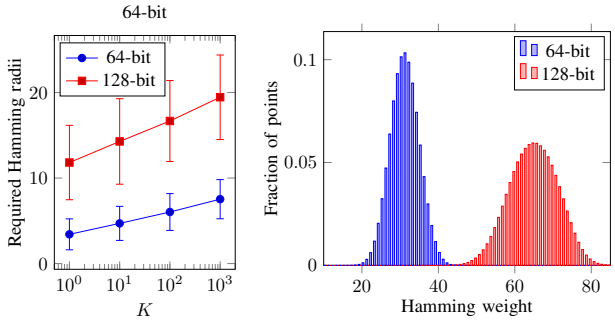


Fig. 2. (a) Average radius of search for solving the $K$NN problem for different values of $K$. (b) Hamming weight histogram of 1 billion SIFT vectors that are mapped to binary space using hyperplane LSH

check a few leaf nodes and such nodes contain a small portion of the dataset points.

However, the pruning power of a depth-one HWT is limited in real applications, mainly due to the fact that the codes are not distributed uniformly among the nodes. While a depth-one HWT can potentially prune the search space of the $r$-neighbor problem and consequently use fewer Hamming distance computations compared to the linear scan, it is only beneficial for small radii of search or very long code lengths. Some problems restrict the search to exact matches [27] or small search radius, but in most cases of interest the desired search radius is large and binary codes are compact. The following two facts limit the performance of a depth-one HWT: 1) Concentration of Hamming weights: since the number of possible binary codes with Hamming weight $c$ is $\binom{p}{c}$, Hamming weights of binary codes (both query and dataset) are highly concentrated around $p/2$. This means that the leaf nodes with Hamming weights around $p/2$ are assigned with a great portion of the points. 2) Large radii of search: solving the $K$NN problem often requires a not-so-small radius and thus we have to check several nodes in such cases. Because of these two observations, we often need to search among several nodes with weights around $p/2$ which unfortunately constitutes a great portion of the codes, thus not much pruning can be done in such cases and the query is virtually compared with all the dataset codes.

To further illustrate this problem in a real application, Figure 2(a) shows the required radius for solving the $K$NN problem with different values of $K$ for a dataset of 1 billion binary codes. Figure 2(b) shows the distribution of Hamming weights for the same dataset which clearly shows the concentration of Hamming weights around $p/2$. As a case in point, to solve the 10NN problem for 64-bit codes, the required search radius is 5 in average. This indicates that to search for

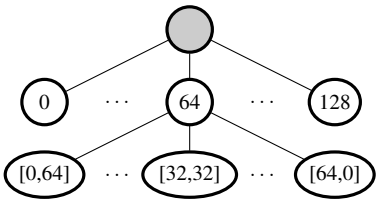


Fig. 3. A HWT with depth 2.

nearest neighbors of a query with $\|\mathbf{q}\|_H = 32$ we have to look among the nodes with Hamming weights $\{27, \ldots, 37\}$ which (based on Figure 2(b)) contain 80% of the points. The problem is that in a vast majority of cases the algorithm requires to compare the query with all of the points in several leaf nodes each of which stores a relatively large number of points. Next, we show that further pruning can be achieved by extending the HWT to higher depths.

### B. Hamming Weight Tree on Substrings

Our approach for enhancing the pruning is to put a limit on the number of binary codes that a leaf node stores. If a node is assigned with more than $\tau$ number of points, it is split by assigning children to the node and moving each binary code to its corresponding child. The children of a node are labeled based on the Hamming weights of *substrings* of the binary codes.

For example, each code that belongs to the node 64 in depth-one tree of Figure 1, can be partitioned into two substrings with equal lengths (the left 64 bits and the right 64 bits). We know that for each code that belongs to this node, the sum of the Hamming weights of the left and right substrings is 64, Therefore, we assign 65 children to this node (see Figure 3), where each child is labeled with one of the possible combinations of Hamming weights for left and right substrings and then move the codes from node 64 to their corresponding children.

In general, each binary code $\mathbf{h} \in \{0,1\}^p$, can be partitioned into $d$ disjoints substrings, $\{\mathbf{h}_d^{(1)}, \ldots, \mathbf{h}_d^{(d)}\}$ each of length $\lfloor p/d \rfloor$ or $\lceil p/d \rceil$. For convenience, in what follows we assume that the substrings contain contiguous bits and $p = 2^t$ and that $p$ is divisible by $d$.

Instead of just considering the Hamming weight of the whole string, we let the tree also incorporate the Hamming weight of the substrings. To that aim, we define the vector transformation $Q_d : \{0,1\}^p \rightarrow \mathbb{N}_0^d$ as follows:

$$Q_d(\mathbf{h}) = [\|\mathbf{h}_d^{(1)}\|_H, \ldots, \|\mathbf{h}_d^{(d)}\|_H], \tag{2}$$

where $\mathbb{N}_0$ denotes the set of non-negative integers. Therefore, $Q_d(\mathbf{h})$ is a vector of length $d$ that contains the Hamming weights of the $\mathbf{h}$'s substrings. We call the output of this transformation the *d-Hamming weight pattern* of $\mathbf{h}$, in which the $i$-th entry denotes the Hamming weight of the $i$-th substring. For example, for the binary code $\mathbf{b} = [1,1,0,0]$, we have $Q_2(\mathbf{b}) = [2,0]$. The insight is that if two binary codes are

close to each other, then their Hamming weight patterns must also be similar.

To measure the similarities between the patterns, one can use the $\ell_p$ norms since patterns lie in a vector space. In particular, we use the $\ell_1$ distance as the measure of similarity between two patterns. Formally, two binary codes $\mathbf{h}$ and $\mathbf{g}$ are said to be $(r, d)$-neighbor pattern of each other if we have:

$$\|Q_d(\mathbf{h}) - Q_d(\mathbf{g})\|_1 \leq r. \tag{3}$$

A special case is when $d = p$ for which we have that $\mathbf{h}$ and $\mathbf{g}$ are $(r, p)$-neighbor pattern of each other if and only if they are $r$-neighbors of each other.

We can now apply (1) to the substrings of binary codes:

*Proposition 2:* If $\|\mathbf{h} - \mathbf{g}\| = \mathbf{r}$, then for any $d < p$ we have:

$$r - \|Q_d(\mathbf{h}) - Q_d(\mathbf{g})\|_1 \in \{0, 2, \ldots, r\}. \tag{4}$$

It is easy to see that proposition 2 is a generalization proposition 1.

Now, reconsider the example of solving the 2-neighbor problem for the query point $\mathbf{q}$, with $\|\mathbf{q}\|_H = 64$, in the HWT shown in Figure 4. As mentioned, only nodes $62,\ldots,66$ can contain such a neighbor. When the algorithm recurses on node 64, it descends down the tree, as it is not a leaf node. Lets assume that for the query code we have that $Q_2(\mathbf{q}) = [32, 32]$. Now, based on (4) it suffices to only search among the nodes [31,33], [33,31], and [32,32] while the remaining 62 children of this node can be ignored. Similarly, if the node [32,32] is later assigned with more than $\tau$ number of points, the algorithm splits it by partitioning each of the two substrings, $\mathbf{h}_2^{(1)}, \mathbf{h}_2^{(2)}$, into four smaller substrings, $\mathbf{h}_4^{(1)}, \mathbf{h}_4^{(2)}, \mathbf{h}_4^{(3)}, \mathbf{h}_4^{(4)}$. Figure 4 shows an example of the paths that must be covered for finding the codes lying at distance $r$ from the query.

Formally, a HWT consists of multiple levels from $-1$ to $l$ for $l \leq \log_2 p$ (for the sake of simplicity in the calculations, we assume that the depth of root is -1). Each binary code of dataset is stored in exactly one leaf node and each node at level $s$ ($s \geq 0$) is labeled with vector $\mathbf{\Phi} = [\phi_1, \ldots, \phi_{2^s}]$ where $\phi_i \in \mathbb{N}_0$. The label of a node specifies the Hamming weight pattern of the codes that belongs to its subtree. In other words, for each code $\mathbf{h}$ that belongs to a node with label $\mathbf{\Phi}$ we have that $\mathbf{\Phi} = Q_d(\mathbf{h})$.

Based on (1), to solve the $r$-neighbor problem at depth $s$ of the tree, the algorithm only needs to recurse on the nodes with labels such as $\mathbf{\Phi} = [\phi_1, \ldots, \phi_{2^s}]$ that satisfy the following equations:

$$\|Q_{2^s}(\mathbf{q}) - \mathbf{\Phi}\|_1 \leq r \tag{5}$$

which is similar to (2). The only difference is that in (5), we are searching for labels of nodes (instead of binary codes) that are $(r, 2^s)$-neighbors of the query. A node at depth $s$ of tree is called a *promising* node if its label is a $(r, 2^s)$-neighbor pattern of the query.

Note that as we descend the tree, more constraints are imposed on the neighbor patterns since the algorithm incorporates piecewise Hamming weights of increasingly finer partitions of the codes. Therefore, not only the Hamming weight of the whole string must be close to the query but also the Hamming weights of the substrings cannot deviated by more than $r$ from those of the query.

In the following, we describe the insert and search operations of HWT in more details.

**Insert.** On the arrival of a new binary code such as $\mathbf{h}$, based on the Hamming weights of $\mathbf{h}$ and its substring, we descend the tree until a leaf node is reached. To descend from a node at level $s$ to a node at level $s + 1$, the $2^{s+1}$-Hamming weight pattern of $\mathbf{h}$ is computed and the child whose label matches the tuple is selected. Therefore, each particular point only participates in one branch of recursion during insertion. Upon reaching a leaf node, the code $\mathbf{h}$ is added to the node if the leaf node is not full. Otherwise, to split a full leaf node at depth $d$, the algorithm computes the $d + 1$-Hamming weight pattern of the binary codes stored at this node, and then moves each of the codes to its corresponding child based on the pattern. Finally, the code $\mathbf{h}$ is similarly added to its corresponding child.

The branching factor of such a node is $(\phi_1 + 1) \times \ldots \times (\phi_w + 1)$. We note that the branching factor can get quite large for deep nodes, however, in high depths, many of children do not store any code. Consequently, instead of initializing all children of a node at once, we use *lazy* initialization to avoid memory allocation for empty children. To do that, rather than storing all children (empty and non-empty), we define an ordering for the children's labels and assign an index to each one. Then, for each node, a hash table is used to store key value pairs where indicies of non-empty children serve as keys, and the values are the pointers to the children. This reduces the storage cost as we only need to store the non-empty children. Meanwhile, insertion, deletion and searching for a child still can be performed in amortized constant time.

**Search.** $r$-neighbor search on a HWT can be answered by proceeding recursively, starting at the root. The search procedure descends through the tree level by level, keeping track of the subset of nodes that may contain the $r$-neighbors of the query. When visiting an internal node, the algorithm only recurses on the children whose label satisfy (5). Thus, starting from the root node, at each depth such as $s$, the search procedure only investigates the non-empty children whose labels are $(r, 2^{s+1})$-neighbor patterns of the query. There are two options for finding the non-empty children that satisfy (5):

- Option 1: The first option is to simply iterate through all children and recurse on those whose pattern satisfy (5).
- Option 2: The second option is to first find the index of all children that satisfy (5) and then recurse on those that exist in the hash table.

For the second approach, the algorithm must enumerate over all promising children of the current node. To that aim, for a node with label vector $[\phi_1, \ldots, \phi_w]$, we find all solutions of

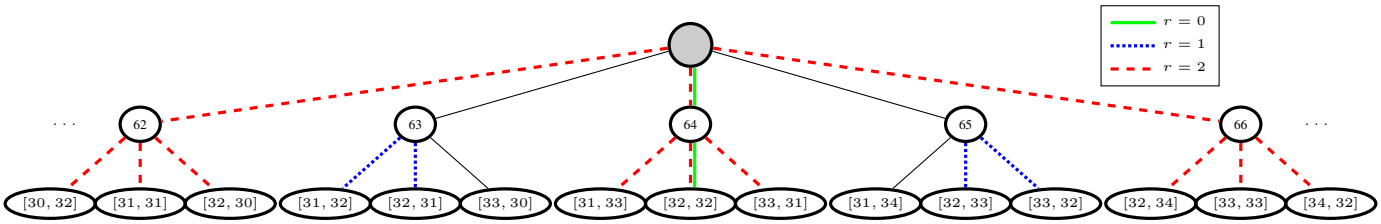


Fig. 4. The paths that must be traversed for finding the codes lying at distance $r$ from the query $\mathbf{q}$ with $\|\mathbf{q}\|_H = 64$, $\|\mathbf{q}_2^{(1)}\|_H = 32$ and $\|\mathbf{q}_2^{(2)}\|_H = 32$. Note that to solve the $r$-neighbor problem, we need to check all nodes lying at distance $r' \leq r$ from the query. For example, to solve the 2-neighbor problem in this tree, we have to traverse all the dashed paths.

the following system of equations:

$$\begin{cases} x_1 + x_2 = \phi_1 \\ \dots \\ x_{2w-1} + x_{2w} = \phi_w \\ \sum_{i=1}^{2w} |\|\mathbf{q}_{2w}^{(i)}\|_H - x_i| \leq r \\ x_i \in \mathbb{N}_0, \end{cases} \tag{6}$$

Each solution vector, $[x_1, \dots, x_{2w}]$, denotes a label of a child that we need to recurse on (provided that it exists in the tree). The equations of the form $x_{2i-1} + x_{2i} = \phi_i$ are necessary to make sure that the solutions are the labels of $j$'s children. The inequality, on the other hand, is necessary to ensure that the solutions are the $r$-neighbor patterns of $\mathbf{q}$.

Not surprisingly, there is a natural trade-off between the two options. For small radii of search and in low depths, the number of solutions to (6) is small and therefore it is perhaps more computationally efficient to use option two. On the other hand, if a node has a small number of children then it is often more efficient to use option 1.

In our implementation, we use the following lower bound on the number of children to decide between the two options:

*Proposition 3:* The number of solutions for (6) is greater than:

$$\sum_{r'=0}^{\lfloor \frac{r}{2} \rfloor} \binom{w + r' - \sum_{i=1}^{w} |\phi_i| - 1}{w - 1}. \tag{7}$$

(Proof in Appendix B).

We use this lower bound such that, at node $j$, if the number of non-empty children is less than (7), then the algorithm proceeds with option 1 otherwise, it proceeds with option 2.

### C. Storage and Computational Costs

We next analyze the storage and computational costs of HWT. Storing the database of binary codes requires $O(np)$ bits. The storage of tree comprises the number of nodes in the tree plus the storage cost of the hash table per node. For each code in a leaf node, we need an identifier that refers to the code in the dataset. This allows one to store the identifier of a code in its corresponding leaf and fetch the full code when necessary. Thus, the total cost of storing identifiers would be $n \log_2 n$. The maximum number of nodes happens when $\tau = 1$ which forms a tree with $n$ leaves in which each leaf stores only one code (provided that there is no duplicate). Assuming that each internal node has at least two children, the number of internal nodes is at most $n - 1$. Therefore, the tree has at most $2n - 1$ nodes. The total cost of hash tables is also bounded by the number of nodes in tree, since each hash table only stores non-empty children. Therefore the storage cost of tree is linear in the number of points.

The storage cost of hash tables depend on the length of keys which in our application represent the index of the nodes. In general, the required length of indices gets longer as we descend in the tree. The number of possible children of a node at a certain depth depends on both the depth itself and the label of node. It is easy to see that for a node at depth $s$, the maximum number of children belongs to the node with pattern $[\frac{p}{2^{s+1}}, \dots, \frac{p}{2^{s+1}}]$, and the number of children for this pattern is:

$$I(s) = \left(\frac{p}{2^{s+1}} + 1\right)^{2^s} \tag{8}$$

Considering the fact that for an internal node we have $s < \log p$, the maximum of $I(s)$ occurs at $s = \log p - 1$. Therefore, assuming $p = 2^t$, the number of bits required to index a child of a node is upper bounded by:

$$\log(\max(I(s))) = 2^{t-1} \log\left(\frac{p}{2^t} + 1\right) = \frac{p}{2}, \tag{9}$$

which is of $O(p)$. Since the number of node indicies in the hash tables is $n$, the total storage complexity of HWT is of $O(np + n \log n) = O(np)$.

Interestingly, the storage cost of HWT is the same as linear scan and better than multi-index hashing technique proposed in [26] and the same as the one in [23].

The insertion time of HWT is also appealing. Starting from the root, at each depth, we just need to compute the pattern of the code at that depth which can done in $O(p)$. Retrieving a pointer to a specific child at a node can be done in amortized $O(1)$ as we are using hash tables. Since the maximum depth of tree is $O(\log p)$, the total cost of inserting a new item is $O(p \log p)$. Finally, each insertion in the worst case can trigger reinsertions of $\tau$ other items but for a fixed $\tau$ the cost is still of $O(p \log p)$.

We also show that, for uniformly distributed binary points, the computational cost of $r$-neighbor search is sublinear in the number data points.

*Theorem 1:* [*Search Complexity for Uniform Data*] *Let $X_n$ be a set of $n$ points, generated independently from the uniform distribution over the $p$-dimensional binary cube $\{0,1\}^p$. Then,*

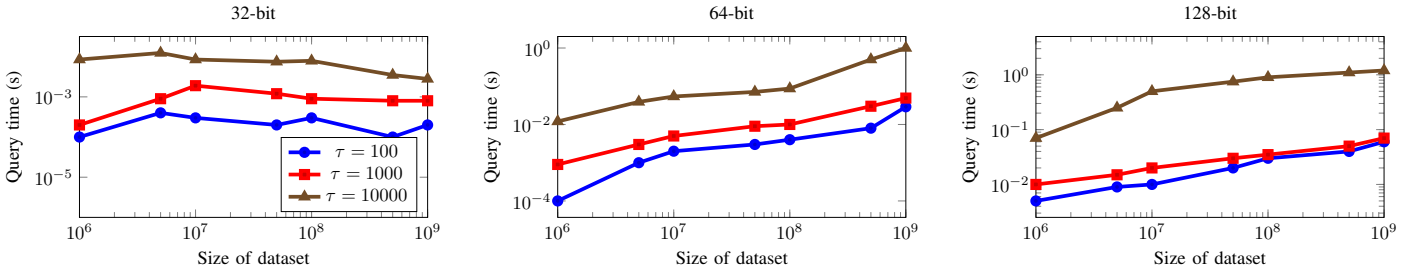


Fig. 5. Average query time of the nearest neighbor search for $\tau$=100, 1000 and 10000 on ANN_1B dataset.

*the expected cost of a single $r$-neighbour search over the Hamming Weight Tree built on $X_n$ is $O(p \log p (\log n)^{4r})$.* (Proof in Appendix D)

Therefore the query cost is sublinear in the number of data points for small radii. The exponential dependency on $r$ can be reduced by incorporating similar ideas used in [26], [28], [29]. The idea is that, given a data structure that solves the $r$-neighbor problem in sublinear time, one can create several such data structures (say $m$ of them) on the substrings of binary codes (in our case it would be a forest of Hamming weight trees with $m$ trees on the substrings). Based on the pigeonhole principle, instead of solving the $r$-neighbor problem on the whole binary code, one can solve $m$ number of $\frac{r}{m}$-neighbor problems, one per substring, and then aggregate the results to retrieve the neighbors. While our current implementation of HWT supports search on multiple trees, we postpone the theoretical and empirical analysis of this idea to future works.

### D. K Nearest Neighbors Search

HWT is inherently designed to answer $r$-neighbor queries. Consequently, each search query to this data structure should contain the query vector and the radius of search, however, the nearest neighbor search does not provide the radius in the query, making it a harder problem than the $r$-neighbor problem. It turns out that for the nearest neighbor of query, the required radius of search for two different query points may vary significantly, depending on how dense the area around the query is populated. Even for a single dataset, the required radius of search for different queries may vary dramatically [30], [23]. If the search radius is set too small, then the algorithm may return no points. On the contrary, large values of radius can result in non-informative neighbors. Moreover, for a large radius of search, the needed time to retrieve the neighbors would be high. Therefore, it is natural for many tasks to fix the number of neighbors and let the radius depend on the query and dataset distribution. Fortunately, a careful implementation of the proposed HWT can be adapted to accommodate the nearest neighbor search queries. Given a binary query point, starting from radius search of zero ($r = 0$), one can progressively increase $r$ until the nearest neighbors are retrieved.

In a naive implementation of HWT, when the radius increases, the new $r$-neighbor search starts from scratch and we have to check all the nodes that may contain the $r$-neighbors in their subtrees. However, many of such nodes overlap with those that are checked for smaller values of $r$. In fact, when $r$ increases, the algorithm have already checked all the nodes that can contain codes with any distance less than $r$ from query. Therefore, we just need to search for the codes that lie at exact distance of $r$ from the query. More specifically, it is easy to see that, all the nodes that must be visited for solving the $r$-neighbor problem, must be also visited for solving the $(r+2)$-neighbor problem (see Figure 4 for $r = 0$ and $r = 2$). To avoid such extra checking, one can store a list of identifiers to the so far internal visited nodes along with their radius of search for which the specific node was visited. Then, to solve the $(r+2)$-neighbor problem, the algorithm can iterate through the list and for each node recuse on those children that may can contain the codes with distance $r+2$ from the query (refer to 4). By doing so, the algorithm can skip many of edge traversals when the radius increases.

## IV. Experimental Results

In this section, we empirically gauge the performance of HWT in comparison with linear scan baseline and MIH. The following experiments are run on a single core 2.0 GHz CPU with 256 GB of RAM. Linear scan and HWT are both coded in C++ and compiled with GCC 4.4.4 using same flags. We used the publicly available implementation of MIH in our experiments.

### A. Datasets

We evaluate the performance of HWT using two well-known real-world datasets which are publicly available: 1) ANN_1B [31] with 1 billion 128D SIFT vectors, and 2) 80 million 384D Gist descriptors from the 80 million tiny images [32]. Each experiment requires two sets of items; base set for populating HWT and the query set that comprises the query points. For 80M Gist descriptors, we randomly select 1000 points to form the query set and use the remaining as the base set. The ANN_1B corpus is already divided into 1 billion base data points and $10^4$ query points from which we randomly select 1000 query points. Therefore, each experiment involves 1000 queries for which the average run-time is reported.

To binarize the datasets, we use the well-known hyperplane LSH [33] which utilizes sign-random projection. More specifically, after zero-centering the data, to encode each bit, first a

random hyperplane is selected where each component of the direction is generated from a normal density, then the value of the bit is specified depending on which side of the hyperplane the point lies. For each dataset, we generate 32-bit, 64-bit and 128-bit binary datasets. With two datasets, and three different code lengths, we obtain 6 binary datasets.

### B. Effect of Threshold Value

We first investigate the effect of parameter $\tau$ on the average query time. This parameter determines the maximum number of binary codes that can be assigned to a leaf node. We have a natural trade-off for different settings of this parameter. Large values of $\tau$ form shallow trees and therefore less pruning takes place which increases the required number of Hamming distance computations. Meanwhile, for each query, fewer node traversals and child checkings are required. In the extreme case, we can create a depth-one tree by setting $\tau$ sufficiently large. On the other hand, small values of $\tau$ cause further pruning and more node traversals.

Figure 5 shows the average query time for different values of $\tau$. The figure indicates that smaller values of $\tau$ results in a faster query time. This shows that further pruning of the search space often results in a better average query time, even with the overhead imposed for finding the promising children of nodes. Nonetheless, since we create a hash table for each internal node, we observed that the memory footprint increases as we decrease $\tau$. Interestingly, this parameter can be set based on the available memory of the target platform to balance the query time and memory requirement.

Note that in some limited cases the query time decreases for larger sizes of dataset. This is mainly due to the fact that increasing the number of points often makes the required search radius for retrieving the nearest neighbor smaller. This in turn lets the tree to search among a fewer number of nodes.

For the following experiments, we choose $\tau = 1000$. For this choice of $\tau$, our current implementation of HWT requires 50 GB, 62 GB, and 73 GB of memory to index 1 billion 32-bit, 64-bit, and 128-bit codes, respectively, which is comparable with that of MIH (refer to [28]).

### C. HWT vs Linear Scan

In this experiment, we focus on comparing the average query time of HWT and linear scan on all datasets. First, we report the average query time when all items are inserted in the tree (batch data), and then we illustrate the query time when the data is inserted sequentially (online data). Table I reports the average query time of the linear scan baseline and HWT along with speed up factors gained by using HWT for different $K$NN problems. For a large range of code lengths and different values of $K$, HWT can achieve orders of magnitude speed up in comparison with the linear scan. Note that the running time of linear scan neither depends on $K$ nor on the underlying distribution of points, however, both factors affect HWT. As $K$ increases, the required search radius also increases which causes longer query time. This is reflected in the reduction of speed up factors when the value of $K$ increases.

| | #bits | Method | $K$ | Time (s) | Speed-up |
|---|---|---|---|---|---|
| ANN_1B | 32 | LS | Any $K$ | 18.14 | 1× |
| | | HWT | 1 | 0.0008 | 22675× |
| | | HWT | 10 | 0.0055 | 3088× |
| | | HWT | 100 | 0.015 | 1029× |
| | 64 | LS | Any $K$ | 22.47 | 1× |
| | | HWT | 1 | 0.049 | 458× |
| | | HWT | 10 | 0.078 | 150× |
| | | HWT | 100 | 0.249 | 90× |
| | 128 | LS | Any $K$ | 32.11 | 1× |
| | | HWT | 1 | 0.07 | 459× |
| | | HWT | 10 | 0.09 | 356× |
| | | HWT | 100 | 0.366 | 88× |
| GIST 80M | 32 | LS | Any $K$ | 1.02 | 1× |
| | | HWT | 1 | 0.003 | 340× |
| | | HWT | 10 | 0.005 | 204× |
| | | HWT | 100 | 0.003 | 340× |
| | 64 | LS | Any $K$ | 1.22 | 1× |
| | | HWT | 1 | 0.009 | 113× |
| | | HWT | 10 | 0.015 | 81× |
| | | HWT | 100 | 0.053 | 23× |
| | 128 | LS | Any $K$ | 2.5 | 1× |
| | | HWT | 1 | 0.08 | 31× |
| | | HWT | 10 | 0.15 | 16× |
| | | HWT | 100 | 0.5 | 5× |

TABLE I
AVERAGE RUNNING TIME FOR A SINGLE NEAREST NEIGHBOR QUERY WITH HWT AND LINEAR SCAN (LS) ALGORITHMS ON TWO DATASETS WITH THREE DIFFERENT CODE LENGTHS.

It is also important to compare the average query time of linear scan and HWT in online setting where the size of the dataset grows gradually. To that end, Figure 6 shows the average query time for the task of $K$NN search for different sizes of dataset and various values of $K$. All plots are shown on log-log axes as the normal axes. It is evident that HWT is faster than linear scan for a wide range of database sizes. Also, the difference between HWT and linear scan grows for larger datasets, making HWT more efficient for large-scale datasets.

### D. HWT vs MIH

We also compare the performance of HWT with MIH which to the best of our knowledge has the best query time for solving the exact HNN problem. To set the number of hash tables for MIH, Norouzi et. al [23] used a hold-out validation set of the database entries. From that set, the running time of the algorithm for different values of $m$ (number of hash tables) is estimated, and the one with the best result is selected. They empirically observed that the optimal value for $m$ is typically close to $p/\log_2 n$. However, in online settings the data points become available sequentially thus the value of $n$ varies over time and the items of dataset are not available in advance. In our experiments, we execute MIH with different values of $m$

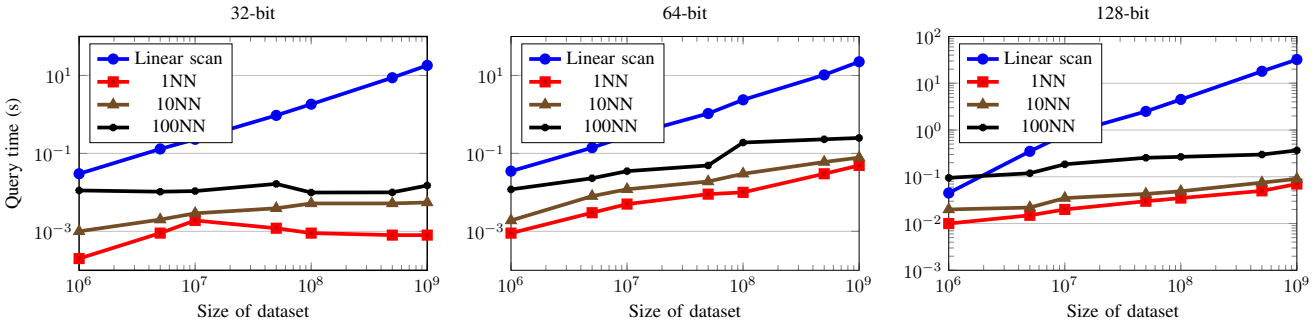


Fig. 6. Average query time of HWT and linear scan on the ANN_1B dataset.

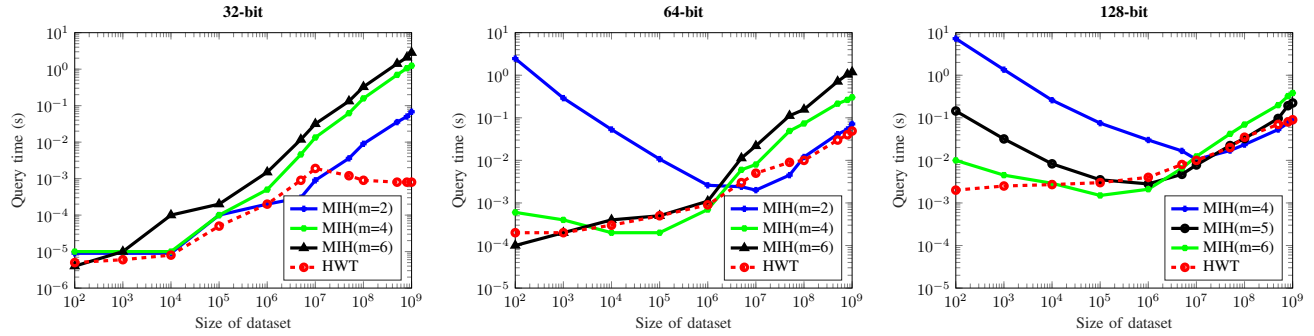


Fig. 7. Average query time of HWT and MIH for the task of nearest neighbor search. The value of $m$ denotes the number of hash tables used in MIH.

for different sizes of the database to investigate the relative performance of these two technique. Not surprisingly, there is a natural trade off between large and small values of $m$. Large values result in assigning fewer number of bits to each table. Therefore, each bucket of hash table can be assigned with several codes and consequently the query must be compared with more codes. On the other hand, small values leads to checking more buckets.

Figure 7 shows the average query time of MIH and HWT for the task of nearest neighbor search (1NN). For MIH, we tried all values of $m \in \{1, \ldots, 10\}$ and measured the average query time (some values resulted in segmentation fault due to high memory overhead) but here we only report those that exhibit better performance than HWT for at least one of the dataset sizes. The figure shows that, for each value of $m$, there is often a range of database size in which MIH outperforms HWT (often when $m$ is close to $p/\log_2 n$). This trend can be seen more clearly in 64-bit and 128-bit codes, nevertheless outside of this range the HWT performs better. Moreover, for large number of codes HWT often exhibit superior performance. Due to the lack of space, the average performance of techniques over different sizes of dataset is omitted from this section, but it indicates that for all lengths of code the average query time of HWT (averaged over different sizes of dataset) is the smallest. In general, MIH in its optimal parameter setting has a marginally better performance but when the number of hash table deviates from its optimal value, HWT outperforms MIH. Therefore, when all binary codes are available at one time, using HWT is not the best choice as compared with the MIH. However, HWT solves a different problem with its own merits.

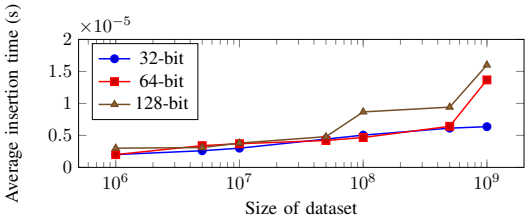


Fig. 8. Average insertion time for different sizes of database and different code lengths.

### E. Insertion Time

One of the key features of the HWT is that it supports insertion of new items. To empirically gauge the performance of the insert procedure, the binary codes of the ANN_1B are sequentially inserted into the tree and the average insertion time for each size is reported in Figure 8. It shows that HWT enjoys a very fast insertion time, making it a suitable fit for dynamic datasets. In particular, the average insertion time is less than 20 microseconds in all of experiments. As the size of the dataset increases, there is a slight growth in the insertion time, mainly due to the collision handling and also rehashing of the elements when the load factor goes beyond a threshold. The total time for inserting the whole 1 billion points was around 2, 3 and 4 hours for 32, 64 and 128 bits codes, respectively.

## V. Conclusion

In this work, we focused on the $K$ nearest neighbors search problem in binary datasets when both the query points and dataset items become available gradually. Based on the branch and bound paradigm, we proposed a tree data structure that solves the nearest neighbor problem much faster than the linear scan and MIH. The proposed data structure constructs a tree over data points in an incremental fashion by routing incoming points to the leaves. We empirically showed that the proposed technique can achieve orders of magnitude speed up in comparison with the linear scan and MIH specially when the size of the dataset is large.

## References


[1] O. Chapelle, P. Haffner, and V. N. Vapnik, "Support vector machines for histogram-based image classification," *IEEE Transactions on Neural Networks*, vol. 10, no. 5, pp. 1055–1064, 1999.

[2] M. Hein and O. Bousquet, "Hilbertian metrics and positive definite kernels on probability measures." in *International Conference on Artificial Intelligence and Statistics*, 2005, pp. 136–143.

[3] M. Calonder, V. Lepetit, C. Strecha, and P. Fua, "Brief: Binary robust independent elementary features," in *European Conference on Computer Vision*. Springer, 2010, pp. 778–792.

[4] S. Leutenegger, M. Chli, and R. Y. Siegwart, "Brisk: Binary robust invariant scalable keypoints," in *IEEE International Conference on Computer Vision*, 2011, pp. 2548–2555.

[5] R. Salakhutdinov and G. Hinton, "Semantic hashing," *International Journal of Approximate Reasoning*, vol. 50, no. 7, pp. 969–978, 2009.

[6] Y. Gong, S. Lazebnik, A. Gordo, and F. Perronnin, "Iterative quantization: A procrustean approach to learning binary codes for large-scale image retrieval," *IEEE Transactions on Pattern Analysis and Machine Intelligence*, vol. 35, no. 12, pp. 2916–2929, 2013.

[7] M. Norouzi and D. M. Blei, "Minimal loss hashing for compact binary codes," in *International Conference on Machine Learning*, 2011, pp. 353–360.

[8] M. Ou, P. Cui, F. Wang, J. Wang, and W. Zhu, "Non-transitive hashing with latent similarity components," in *International Conference on Knowledge Discovery and Data Mining*, 2015, pp. 895–904.

[9] R. Raziperchikolaei and M. A. Carreira-Perpinán, "Learning independent, diverse binary hash functions: Pruning and locality," in *IEEE 16th International Conference on Data Mining*. IEEE, 2016, pp. 1173–1178.

[10] Y. Weiss, A. Torralba, and R. Fergus, "Spectral hashing," in *Advances in Neural Information Processing Systems*, 2009, pp. 1753–1760.

[11] E.-J. Ong and M. Bober, "Improved hamming distance search using variable length substrings," in *IEEE Conference on Computer Vision and Pattern Recognition*, 2016, pp. 2000–2008.

[12] L.-K. Huang, Q. Yang, and W.-S. Zheng, "Online hashing," *International Joint Conference on Artificial Intelligence*, pp. 1442–1428, 2013.

[13] L. K. Huang, Q. Yang, and W. S. Zheng, "Online hashing," *IEEE Transactions on Neural Networks and Learning Systems*, 2017.

[14] M. Minsky and S. Papert, *Perceptrons.* MIT press, 1969.

[15] H. Liu, R. Wang, S. Shan, and X. Chen, "Deep supervised hashing for fast image retrieval," in *IEEE Conference on Computer Vision and Pattern Recognition*, June 2016.

[16] Y. Ke, R. Sukthankar, L. Huston, Y. Ke, and R. Sukthankar, "Efficient near-duplicate detection and sub-image retrieval," in *ACM Multimedia*, vol. 4. Citeseer, 2004, p. 5.

[17] H. Jegou, M. Douze, and C. Schmid, "Hamming embedding and weak geometric consistency for large scale image search," in *European Conference on Computer Vision*. Springer, 2008, pp. 304–317.

[18] O. Barkol and Y. Rabani, "Tighter bounds for nearest neighbor search and related problems in the cell probe model," in *Annual ACM symposium on Theory of computing*, 2000, pp. 388–396.

[19] J. Alman and R. Williams, "Probabilistic polynomials and hamming nearest neighbors," in *Annual Symposium on Foundations of Computer Science*, 2015, pp. 136–150.

[20] P. Indyk and R. Motwani, "Approximate nearest neighbors: towards removing the curse of dimensionality," in *Annual ACM symposium on Theory of computing*. ACM, 1998, pp. 604–613.

[21] A. Beygelzimer, S. Kakade, and J. Langford, "Cover trees for nearest neighbor," in *ACM International Conference on Machine Learning*, 2006, pp. 97–104.

[22] M. Muja and D. G. Lowe, "Scalable nearest neighbor algorithms for high dimensional data," *IEEE Transactions on Pattern Analysis and Machine Intelligence*, vol. 36, no. 11, pp. 2227–2240, 2014.

[23] M. Norouzi, A. Punjani, and D. J. Fleet, "Fast search in hamming space with multi-index hashing," in *IEEE Conference on Computer Vision and Pattern Recognition*, 2012, pp. 3108–3115.

[24] H. Samet, *Foundations of multidimensional and metric data structures*. Morgan Kaufmann, 2006.

[25] A. Torralba, R. Fergus, and Y. Weiss, "Small codes and large image databases for recognition," in *Computer Vision and Pattern Recognition*. IEEE, 2008, pp. 1–8.

[26] D. Greene, M. Parnas, and F. Yao, "Multi-index hashing for information retrieval," in *Annual Symposium on Foundations of Computer Science*. IEEE, 1994, pp. 722–731.

[27] M. Muja and D. G. Lowe, "Fast matching of binary features," in *Conference on Computer and Robot Vision*. IEEE, 2012, pp. 404–410.

[28] M. Norouzi, A. Punjani, and D. J. Fleet, "Fast exact search in hamming space with multi-index hashing," *IEEE Transactions on Pattern Analysis and Machine Intelligence*, vol. 36, no. 6, pp. 1107–1119, 2014.

[29] X. Zhang, J. Qin, W. Wang, Y. Sun, and J. Lu, "Hmsearch: An efficient hamming distance query processing algorithm," in *International Conference on Scientific and Statistical Database Management*. ACM, 2013, p. 19.

[30] J. Gao, H. Jagadish, B. C. Ooi, and S. Wang, "Selective hashing: Closing the gap between radius search and k-nn search," in *International Conference on Knowledge Discovery and Data Mining*, ser. KDD '15. ACM, 2015, pp. 349–358.

[31] H. Jégou, R. Tavenard, M. Douze, and L. Amsaleg, "Searching in one billion vectors: re-rank with source coding," in *IEEE International Conference on Acoustics, Speech and Signal Processing*, 2011, pp. 861–864.

[32] A. Torralba, R. Fergus, and W. T. Freeman, "80 million tiny images: A large data set for nonparametric object and scene recognition," *IEEE Transactions on Pattern Analysis and Machine Intelligence*, vol. 30, no. 11, pp. 1958–1970, 2008.

[33] M. S. Charikar, "Similarity estimation techniques from rounding algorithms," in *ACM Symposium on Theory of Computing*. ACM, 2002, pp. 380–388.


## Appendix

### A. Proof of proposition 1

By definition of Hamming distance, $r$ bits of $\mathbf{h}$ are flipped in $\mathbf{g}$. Such flips can be of types 1) zero to one, or 2) one to zero. Let $r_1$, $r_2$ denote the number of type 1 and type 2 flips respectively, thus we have $r_1 + r_2 = r$. We have:

$$\|\mathbf{h}\|_H + r_1 - r_2 = \|\mathbf{g}\|_H. \tag{10}$$

Using $r_1 + r_2 = r$, we have:

$$\|\mathbf{h}\|_H - \|\mathbf{g}\|_H = 2r_2 - r. \tag{11}$$

The proof is concluded considering the fact that $r_2 \leq r$

### B. Proof of proposition 3

The unknowns of (6) come in pairs, tied together only by the last inequality. To solve it, one can consider $r+1$ systems of equations, one for each possible value of $\sum_{i=1}^{2w} |\|\mathbf{q}_{2w}^{(i)}\|_H - x_i| = r', r' \in \{0, \dots r\}$ , and solve them independently.

Now consider one of such systems in which $\sum_{i=1}^{2w} \|\mathbf{q}_{2w}^{(i)}\|_H - x_i| = r'$. Given $r'_k$ and $\phi_k$ $(0 \leq k \leq w)$, the following system of equations can be solved easily:

$$\begin{cases} x_{2k-1} + x_{2k} = \phi_k \\ |\|\mathbf{q}_{2w}^{(2k-1)}\|_H - x_{2k-1}| + |\|\mathbf{q}_{2w}^{(2k)}\|_H - x_{2k}| = r'_k. \end{cases} \tag{12}$$

which has a set of solutions only if i) $r'_k \geq |\phi_k|$ and ii) $r'_k \equiv \phi_k \bmod 2$. For each $k$, it can be solved by considering four cases according to whether $x_{2k-1}$ and $x_{2k}$ are positive or negative. To recover all solutions for a specific value of $r'$, we need to iterate through all possible values of $r'_k$s and for each iteration we must solve an instance of (12). In other words, we need to generate all possible distributions of $r'$ among the $w$ equations. The total number of distributions for a specific $r'$ is essentially the number of partitions of parameter $r'$ into $w$ non-negative integers, $r_1 + \ldots + r_w = r'$, given by the *multiset coefficient*:

$$\left(\!\binom{r'+1}{w-1}\!\right) \triangleq \binom{w+r'-1}{w-1}. \tag{13}$$

Thus, the total number of instances of (12) that we need to solve is:

$$\sum_{r'=0}^{r} \left(\!\binom{r'+1}{w-1}\!\right). \tag{14}$$

However, many of the instances do not yield a solution as they can violate constraints (i) or (ii). What we show is that we can skip those instances with a simple approach. To skip the instances that violate constraint (i), we can first find all partitions of $r_1 + \ldots + r_w = r' - \sum_{i=1}^{w} |\phi_i|$ and then for each partition add the $|\phi_i|$ to its corresponding $r_i$. This reduces the number of instances to $\sum_{r'=0}^{r} \left(\!\binom{r'+1-\sum_{i=1}^{w}\phi_i}{w-1}\!\right)$.

Now, to also ensure that all partitions satisfy constraint (ii), we can limit the radius to $r/2$ which would result in (7). In simple terms, instead of finding all $\sum_{r'=0}^{r} \left(\!\binom{r'+1-\sum_{i=1}^{w}\phi_i}{w-1}\!\right)$ solutions and then discarding those not satisfying the $r'_k \equiv \phi_k \bmod 2$ condition, one can first find all solutions of $r'_1 + \ldots + r'_w = r'/2 - \sum_{i=1}^{w} |\phi_i|$ and then for each solution multiply all of the $r'_i$s ($1 \leq i \leq w$) by two. Finally, the entries that must be odd are added by one. Going from (14) to (7) saves us a lot of unnecessary computation as the first quantity is much bigger.

Using this approach, each generated instance of (12) has at least one solution which directly translate to that (7) is a lower bound for the number of children that must be checked.

### C. Proof of proposition 2

If $\|\mathbf{h}-\mathbf{g}\|_H = r$, we have that $\sum_{i=1}^{d} \|\mathbf{h}_d^{(i)} - \mathbf{g}_d^{(i)}\| = r$. Due to proposition 1, for the $i$-th substring, $1 \leq i \leq d$, we have:

$$r_i - |\|\mathbf{h}_d^{(i)}\|_H - \|\mathbf{g}_d^{(i)}\|_H| \in \{0, 2, \ldots, r_i\} \tag{15}$$

where $r_i = \|\mathbf{h}_d^{(i)} - \mathbf{g}_d^{(i)}\|_H$. Summing over all substrings would result in:

$$\sum_{i=1}^{d} r_i - \sum_{i=1}^{d} |\|\mathbf{h}_d^{(i)}\|_H - \|\mathbf{g}_d^{(i)}\|_H| \in \{0, 2, \ldots, \sum_{i=1}^{d} r_i\}$$

The proof is completed considering the fact that $r = \sum_{i=1}^{d} r_i$ and $\|Q_d(\mathbf{h}) - Q_d(\mathbf{g})\|_1 = \sum_{i=1}^{d} |\|\mathbf{h}_d^{(i)}\|_H - \|\mathbf{g}_d^{(i)}\|_H|$

### D. Proof of Theorem 1

We start the proof of Theorem 1 by defining the event of *collision* over $d$-patterns, and then provide an upper bound on the probability of such collision.

*Definition 1 (Collision):* Let $X_n = \{\mathbf{x}^i\}_{i=1}^n$ be a set of $p$-dimensional binary vectors, i.e., $\mathbf{x}^i \in \{0,1\}^p$. We say that query $\mathbf{q} \in \{0,1\}^p$ $d$-collides with $X_n$ if

$$\exists i \in [n], \text{ s.t. } Q_d(\mathbf{x}^i) = Q_d(\mathbf{q})$$

*Lemma 1:* Let $X_n = \{\mathbf{x}^i\}_{i=1}^n$ and $\mathbf{q}$ be $n+1$ iid random variables generated from the uniform distribution over the $p$-dimensional binary cube. Then, for every $0 < d \leq p$, the probability that $\mathbf{q}$ $d$-collides with $X_n$ is at most $n(\frac{d}{p})^{\frac{d}{2}}$.

*proof.* Note that because $\mathbf{q}$ is uniformly generated, the distribution of its Hamming weight (i.e., distribution of $\|\mathbf{q}\|_H$) is binomial. Similarly, the distribution of the Hamming weight of any substring of $\mathbf{q}$ is binomial, i.e., $\|\mathbf{q}_d\|_H \sim Bin(\frac{p}{d}, 1/2)$. Here, $Bin(n,p)$ denotes a binomial pdf with parameters $n$ (number of draws) and $p$ (probability of success). Now with an application of union bound, and the fact that the $d$ different patterns are generated independently we have:

$$\begin{aligned}
&Pr\left[\exists i \in [n] \text{ s.t. } Q_d(\mathbf{x}^i) = Q_d(\mathbf{q})\right] \leq nPr\left[Q_d(\mathbf{x}^1) = Q_d(\mathbf{q})\right] \\
&\leq n\left(Pr\left[\|\mathbf{x}_d^{1(1)}\|_H = \|\mathbf{q}_d\|_H\right]\right)^d \\
&\leq n\left(\sum_{j=0}^{\frac{p}{d}} Pr\left[\|\mathbf{x}_d^{1(1)}\|_H = \|\mathbf{q}_d^{(1)}\|_H = j\right]\right)^d \\
&\leq n\left(\sum_{j=0}^{\frac{p}{d}} Pr\left[\|\mathbf{x}_d^{1(1)}\|_H = j\right] Pr\left[\|\mathbf{q}_d^{(1)}\|_H = j\right]\right)^d \\
&\leq n\left(\sum_{j=0}^{\frac{p}{d}} Bin(\frac{p}{d}, 1/2)\Big|_j \, Bin(\frac{p}{d}, 1/2)\Big|_j\right)^d \\
&\leq n\left(\sum_{j=0}^{\frac{p}{d}} Bin(\frac{p}{d}, 1/2)\Big|_j \, Bin(\frac{p}{d}, 1/2)\Big|_{\frac{p}{2d}}\right)^d \\
&\leq n\left(\sqrt{\frac{d}{p}} \sum_{j=0}^{\frac{p}{d}} Bin(\frac{p}{d}, 1/2)\Big|_j\right)^d \leq n\left(\frac{d}{p}\right)^{\frac{d}{2}}
\end{aligned}$$

where we used the fact that the binomial pmf, $Bin(\frac{p}{d}, 1/2)|_j$, is maximized when $j = \frac{p}{2d}$.

Now we are ready to prove Theorem 1. We first bound the computational cost of search associated with a single layer, and then aggregate the cost over all of the layers.

In depth $s$, each node corresponds to a $d$-pattern where $d = 2^s$. Also, for any query $\mathbf{q}$, the number of $(r,d)$-neighbors patterns ($r$-vicinity) for $\mathbf{q}$ is at most $r\binom{d+r-1}{r} \leq d^r$. In other words, in layer $s$, there are at most $d^r = 2^{rs}$ different potential nodes that are the $(r,d)$-neighbor patterns of $\mathbf{q}$. The critical observation is that any operation in layer $s$ is performed on a subset of these nodes, but not all of these potential nodes are actually materialized.

In fact, a node is accessed only if it is (i) non-empty, and (ii) $d$-collides with a point in $r$-vicinity of $\mathbf{q}$. But based on Lemma 1, the expected number of such nodes—i.e., non-empty nodes in layer $s$ that $d$-collide with a point in $r$-vicinity of $\mathbf{q}$—is bounded by $2^{rs} \min\left(1, n(\frac{2^s}{p})^{2^{s-1}}\right)$. For each of these nodes, we check at most $2^{rs}$ potential solutions to Equation 6. Therefore, in layer $s$, in total we would have at most $2^{rs}2^{rs} \min\left(1, n(\frac{2^s}{p})^{2^{s-1}}\right)$ many patterns to check.

The cost of checking whether two $d$-dimensional binary vectors have the same $d$-pattern is $O(p)$. Hence, the total cost associated with layer $s$ is $O\left(p2^{2rs} \min\left(1, n(\frac{2^s}{p})^{2^{s-1}}\right)\right)$. Finally, we have at most $\log p$ layers, so the total cost is $O\left(2^{2rs} \min\left(1, n(\frac{2^s}{p})^{2^{s-1}}\right) p \log p\right)$, which we claim is in fact $O\left(p \log p (\log n)^{4r}\right)$. This is clear when $s < 1+\log\log n$. Also, if $s \geq 1 + \log\log n$, we can assume $p > \frac{s}{2}$ (because the last layer will not have any children) so $2^{2rs} \min\left(1, n(\frac{2^s}{p})^{2^{s-1}}\right) \leq (\log n)^4$ which completes the proof.