Online Nearest Neighbor Search Using Hamming Weight Trees

Sepehr Eghbali, Hassan Ashtiani, and Ladan Tahvildari

Abstract—Nearest neighbor search is a basic and recurring proximity problem that has been studied for several decades. The goal is to preprocess a dataset of points so that we can quickly report the closet point(s) to any query point. Many recent applications of NNS involve datasets that are very large and dynamic, that is items of data items become available gradually. In this study, we propose a data structure for solving NNS for dynamic binary data where both query and dataset items are represented as binary strings. The proposed tree data structure, called the Hamming Weight Tree, is simple and as the names suggests, is based on partitioning the feature space of binary strings by exploiting the Hamming weights of the binary codes and their substrings. Given a Hamming Weight Tree of binary codes, we propose two search algorithms that accommodate nearest neighbor search for two different distance functions, the Hamming distance and the angular distance. Our empirical results show significant speedup in comparison with the best known large-scale solutions.

Index Terms—Nearest neighbor search, binary codes, sublinear search, tree search, Hamming Weight

1 Introduction

Nearest Neighbor Search (NNS) is a fundamental problem which arises as the core component of many machine vision and pattern recognition techniques such as image search [1], [2], matching local features [3], object segmentation [4] and parameter estimation [5]. It is also a source of many interesting theoretical developments [6]. In its typical setting, the NNS problem is defined as follows. Given a (static or dynamic) dataset of points from a metric space, the goal is to preprocess the dataset in such a way that, when later given a target vector **q**, one can efficiently find the point in the dataset which is the closest to **q**. For many machine learning tasks on large-scale and high dimensional datasets, NNS can comfortably become the computational bottleneck. Not surprisingly, a great deal of endeavor has been concentrated on designing data structures and algorithms that accommodate fast and scalable search for different variations of NNS problem.

Although many papers have studied NNS for the general euclidean space, only a few have focused on the setting in which items of dataset and the query are represented with binary strings (codes) [7], [8], [9], [10]. Literature abounds with the applications of binary datasets. A typical example is encoding text documents with *term vectors*, where absence and presence of words (or shingles) are captured with binary variables [11], [12]. More recently, in machine vision, a number of hand-crafted binary feature extractors such as BRIEF [13], ORB [14] and BRISK [15] have been used to describe images. The advantage of such lightweight visual binary descriptors over more established descriptors such as SIFT [16] or rich features learned from convolutional networks is that binary features are cheaper to compute, more compact to store and faster to compare.

Nevertheless, perhaps the most notable application of binary codes is *binary hashing* [17] which has been the subject of intense study for more than a decade. The shared goal of binary hashing techniques is to learn a hash function that encodes high dimensional real-valued vectors with *compact* binary codes subject to preserving a notion of similarity. Compact codes are in essence designed to expedite nearest neighbor search in high-dimensional large-scale datasets as they facilitate storage and dramatically reduce complexity of distance computation. Learning good hash functions that better respect the notion of similarity has been an active field of research over the recent decade, resulting in a rich set of binary hashing techniques [18], [19], [20], [21], [22], [23], [24], [25], [26], [27]. The general approach is to design an affinity-based loss function that relates the distance between binary vectors to the supervised neighborhood information in a training set. Once the mapping is learned, the dataset items and the query points are mapped to binary codes and the nearest neighbor search in the original space is now approximated with the one in the target space. At this stage, linear scan is often used to find the nearest neighbor(s) to the query. It is not surprising that due to extremely fast hardware-level bit manipulation functions, performing linear search in the binary space is much faster than in the real space. However, for large datasets, the search time for a single query can still take several minutes [28].

- *S. Eghbali and L. Tahvildari are with the Department of Electrical and Computer Engineering, University of Waterloo, 200 University Ave West, Waterloo, ON N2L 3G1, Canada. E-mail: {s2eghbal, ladan.tahvildari}@uwaterloo.ca.*
- *H. Ashtiani is with the Department of Computing and Software, McMaster University, 1280 Main St W, Hamilton, Hamilton, ON L8S 4L8, Canada. E-mail: zokaeiam@mcmaster.ca.*

Manuscript received 21 June 2018; revised 1 Jan. 2019; accepted 10 Feb. 2019. Date of publication 1 Mar. 2019; date of current version 3 June 2020. (Corresponding author: Sepehr Eghbali.) Recommended for acceptance by J. Zhou. Digital Object Identifier no. 10.1109/TPAMI.2019.2902391

Modern real-life datasets are not only large in the number of points, but also are open-ended and dynamic; new items appear 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 NNS 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 KNN search in dynamic binary datasets?*

This question has not been answered adequately in the literature. Some recent studies have addressed the problem of learning compact binary codes in online settings [29], [30]. They have shown that it is possible to gradually update the hash function (that maps real data to binary codes), as data points become available, such that the resulting binary codes better preserve the similarities. However, the problem of efficiently searching among the so-far collected binary codes seems to have remained unsettled. In practice, researchers often resort to linear scan to find nearest neighbors in online applications.

This paper revisits the exact nearest neighbor search for compact binary codes. We propose a data structure, called *Hamming Weight Tree* (HWT), that enables fast and exact nearest neighbor search under two different distance functions, namely *Hamming* distance and *angular* distance (*cosine similarity*). Importantly, HWT supports insertion of new items which is imperative for dealing with dynamic datasets. Our empirical experiments show that HWT achieves orders of magnitude speedup in comparison with linear scan and outperforms several best known solutions for the static setting.

2 Background Review

The nearest neighbor search problem in the Hamming space, originally presented by Minsky and Papert [31], has been extensively studied in the literature both because of its theoretical importance and because of its numerous applications that abound in image retrieval [19], [32], duplicate detection [33] and matching local features [3]. 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 [34]. Unfortunately, to this day, there is no algorithm with polynomial pre-processing and storage costs which guarantees sublinear query time [35].

To overcome the apparent difficulty of finding the exact solutions, recent studies are mainly focused on retreiving *approximate* neighbors in order to trade accuracy for scalability [36]. The key idea shared by these algorithms is to find the nearest neighbors with high probability, instead of probability one. There are mainly two directions of research in approximate nearest neighbor search (ANN): (1) reducing distance computation cost by embedding the points into a similarity-preserving space where distances can be computed efficiently, (2) reducing the number of distance computations between query and dataset items using space partitioning.

Dimensionality reduction techniques such as random projection [37] and PCA can map the points into a lower dimension while preserving the distances up to an approximation factor. Similarly, binary hashing techniques aim at encoding input vectors with short binary codes that are faithful to a notation of similarity. However, even with the dimensionality reduction, the query point must still be compared with all dataset items, resulting in a linear dependency of query time on the number of points. To achieve sublinear query time, space partitioning algorithms such as Locality Sensitive Haashing (LSH) and tree based indexing divide the input space into multiple cells and then compare the query with the points colliding with cell where the query point belongs to. By adopting randomized space partitioning, LSH managed to break the linear query time bottleneck and had enormous impact. Still, such hashing-based approximate techniques suffer from a less sever problem known as the *curse of approximation* [35], [38], that is either their query or space costs have exponential dependency on the search accuracy

From the practical perspective, 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 [39], [40], [41]. From this perspective, several studies have investigated practical solutions for the NNS in the static setting. Space-partitioning algorithms such as KD tree [42] and Voronoi diagram [43] are among the best known nearest neighbor algorithms for low dimensional spaces (up to 20 or 30) [44]. 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 [39].

When points live in the Hamming 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 [2]. 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 number of data points. This issue turns linear scan into a faster alternative.

Multi-Index hashing (MIH) [28], [41], [45] is a rigorous approach for handling this issue. The key idea of MIH is that two close binary strings must also have close 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 the search is performed in each hash table independently. Empirical experiments show that MIH can provide dramatic speed-up over the linear scan and currently offers state-of-the-art performance in solving nearest neighbor search for Hamming [41] and angular distances [45]. Recently, Ong and Bober [28] proposed an extension MIH to support variable length hashkeys for different hash tables.

Despite its success, the performance of MIH heavily depends on knowing the size of dataset beforehand. In [41] and [46], mpirical 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

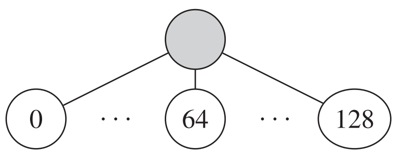


Fig. 1. Hamming weight tree with depth one for 128-bit codes.

work, even significantly more than what linear scan requires. Consequently, MIH is mostly applicable for batch data in which the number of items remains constant and known.

3 Hamming Weight Tree

We start off describing our data structure by focusing on the Hamming NNS problem, meaning that binary codes are compared in terms Hamming distances. Then, in Section 4, we show how the same proposed data structure can be applied to the angular NNS by modifying the search algorithm.

We address two closely related problems. Given a dataset of p-dimensional binary codes $\mathcal{B} = \{\mathbf{b}^i \in \{0,1\}^p\}_{i=1}^n$, and a binary query vector $\mathbf{q} \in \{0,1\}^p$, 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.

3.1 Depth One Tree

We first propose a data structure for solving the r-neighbor problem and then apply the data structure to solve the KNN problem. One of key ideas of this paper rests on the following proposition which intuitively states that 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 nonzero entries in the code.

Proposition 1 ([47]). *If* $\|\mathbf{h}-\mathbf{g}\|_H = r$, *then we have:*

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

where $\|.\|_H$ *denotes the Hamming weight.*

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. 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 $\{\|\mathbf{q}\|_H - r, \dots, \|\mathbf{q}\|_H + r\}$ 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 Fig. 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 a depth-one 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. Pleasingly, 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 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 [48] 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 KNN 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 constitute 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, Fig. 2a shows the required radius for solving the KNN problem in the Hamming space with different values of K for a dataset of 1 billion binary codes. Fig. 2b shows the distribution of Hamming weights for the same dataset which clearly shows the concentration of Hamming weights around $p/2$. For instance, to solve the 10NN problem for 64-bit codes, the required search radius is 5 in average. This indicates that to search for nearest neighbors of a query with $\|\mathbf{q}\|_H = 32$ we have to look among the nodes with Hamming weights $\{27, \dots, 37\}$ which (based on Fig. 2b) contain 80 percent 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 storing a relatively large number of points.

3.2 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

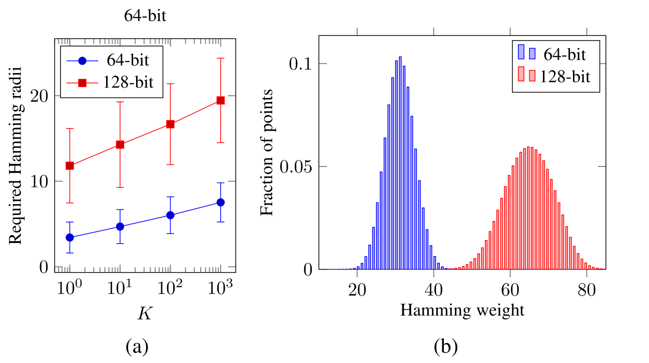


Fig. 2. (a) Average radius of search for solving the KNN problem for different values of K. (b) Average Hamming weight of 1 billion SIFT vectors that are mapped to binary space using hyperplane LSH

assigned with more than τ number of points, it is split by creating multiple children 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 Fig. 1, can be partitioned into two substrings with equal lengths (the left and 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 create 65 children for this node (see Fig. 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, $p = 2^t$, and 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 $Q_d : \{0,1\}^p \to \mathbb{N}_0^d$ 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 with entries denoting 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 ith entry denotes the Hamming weight of the ith 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.

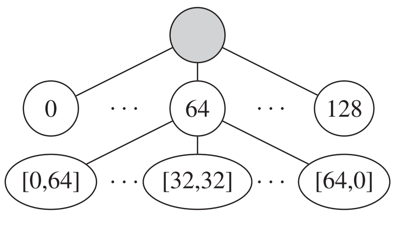


Fig. 3. A possible configuration of Hamming weight tree with depth 2 for 128-bit binary codes.

To measure the similarities between the patterns, one can use the ℓ_p norms since patterns lie in a vector space. In particular, we use the ℓ_1 distance as the measure of similarity between two patterns which corresponds to the sum of the Hamming weight differences. 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. Formally, if $\|\mathbf{h} - \mathbf{g}\|_H = r$, we have that $\|\mathbf{h}_d^{(1)} - \mathbf{g}_d^{(1)}\|_H + \ldots + \|\mathbf{h}_d^{(d)} - \mathbf{g}_d^{(d)}\|_H = r$, therefore by applying proposition 1 to each of the substrings we have:

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

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 Fig. 4. As mentioned, only nodes 62,..., 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 τ number of points, the algorithm splits it by partitioning the two substrings, $\mathbf{h}_2^{(1)}, \mathbf{h}_2^{(2)}$, into four smaller substrings, $\mathbf{h}_4^{(1)}, \mathbf{h}_4^{(2)}, \mathbf{h}_4^{(3)}, \mathbf{h}_4^{(4)}$. Fig. 4 shows an example of the nodes that must be visited 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 $\mathbf{\Phi} = [\phi_1, \ldots, \phi_w]$ where $\phi_i \in \mathbb{N}_0$ and $w = 2^s$. The label of a node specifies the Hamming weight pattern of the codes that belong to its subtree.

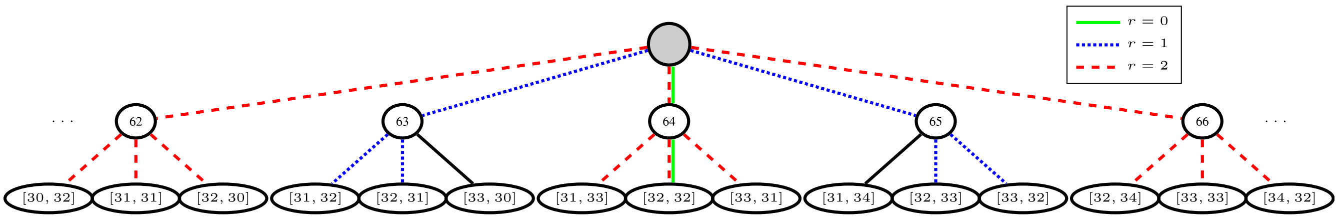


Fig. 4. The paths that must be traversed for finding the codes lying at distance r from the query $\mathbf{q}$ with $\|\mathbf{q}\|_H = 64$, $\|\mathbf{q}_2^{(1)}\|_H = 32$ and $\|\mathbf{q}_2^{(2)}\|_H = 32$. Note that to solve the r-neighbor problem, we need to check all nodes lying at distance $r'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

In other words, for each code $\mathbf{h}$ that belongs to a node $\mathbf{\Phi}$ 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 $\mathbf{\Phi} = [\phi_1, \ldots, \phi_w]$ 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, w)-neighbors of the query. A node at depth s is called a *promising* node if its label is a $(r, 2^s)$-neighbor pattern of query.

Note that as we descend the tree, more constrains 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 deviate 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 s, the algorithm computes the 2^{s+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 an internal node at depth s is $(\phi_1 + 1) \times \ldots \times (\phi_{2^s} + 1)$. Although the branching factor can get quite large for deep nodes, in high depths, most 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 dynamic 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: Simply iterate through all children and recurse on those whose pattern satisfy (5).
- Option 2: 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, at node j labeled $[\phi_1, \ldots, \phi_w]$, we find all solutions of the following system of equations:

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

Each solution vector, $[x_1, \ldots, x_{2w}]$, 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 $\mathbf{q}$. [47] describes an efficient algorithm for solving (6).

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 computationally more 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 2 ([47]). *The number of solutions for (6) is greater than:*

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

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.

3.3 Storage and Computational Costs

We next analyze the storage and computational costs of HWT. Storing the dataset of binary codes requires $O(np)$ bits. The storage cost of the 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 number of nodes in the 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 $[\frac{p}{2^{s+1}}, \ldots, \frac{p}{2^{s+1}}]$, 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:

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

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 [46] (with cost of $O(rn^{1+r/p}\log n)$ for solving r-neighbor search) and the same as those in [9], [45].

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 τ other items but for a fixed τ 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 logarithmic 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, 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. See supplementary material. □

Therefore, the query cost is logarithmic in the number of data points for small radii. The exponential dependency on r can be reduced by drawing upon similar techniques used in [9], [10], [46]. 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, theoretical and empirical analysis of this idea is out of scope of this paper and we postpone it to future works.

3.4 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 [41], [49]. 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 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 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 Fig. 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 iterates through the list and for each node recurse on children that may 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.

4 Angular Nearest Neighbor Search

Although most of studies on the binary codes adopt Hamming distance as the measure of similarity between binary codes, in some applications the angle between the code arises as a more effective alternative [20], [50], [51]. For example, Angular Quantization-based Binary Codes (AQBC) [20] is a binary hashing technique in which the appropriate similarity measure is the cosine of the angle between the binary codes. In its basic form, AQBC maps real valued feature vectors onto the vertex of binary hypercube with which it has the smallest angle. The distance between the resulting codes are then defined as the cosine of angle between them. Recall that the Hamming NNS search on HWT was performed by solving multiple instance of r-neighbor query. Pleasingly, HWT can be used to carry out angular KNN without modifying the data structure and again by only solving r-neighbor queries during the query phase. The only difference from the Hamming distance case is that for the angular case the search executes r-neighbor queries on some particular nodes and in an altered order discussed in the following.

The cosine of the angle between two binary vectors can be computed as follows:

$$sim(\mathbf{q}, \mathbf{b}) = \cos(\theta(\mathbf{q}, \mathbf{b})) = \frac{\mathbf{q}^T\mathbf{b}}{\|\mathbf{q}\|_2\|\mathbf{b}\|_2}. \tag{10}$$

Computing the cosine similarity between two binary codes is marginally slower than computing Hamming distance, however it is still fast compared to computing similarity of the original vectors. As shown in [45], for binary vectors, we can rewrite (10) with:

$$sim(\mathbf{q}, \mathbf{b}) = \frac{\|\mathbf{q}\|_1 - r_1(\mathbf{q}, \mathbf{b})}{\sqrt{\|\mathbf{q}\|_1} \times \sqrt{\|\mathbf{q}\|_1 - r_1(\mathbf{q}, \mathbf{b}) + r_2(\mathbf{q}, \mathbf{b})}}, \tag{11}$$

where $r_1(\mathbf{q}, \mathbf{b}) \triangleq \|\mathbf{q} \wedge \neg\mathbf{b}\|_H$ denotes the number of bits that are 1 in $\mathbf{q}$ and 0 in $\mathbf{b}$ and $r_2(\mathbf{q}, \mathbf{b}) \triangleq \|\neg\mathbf{q} \wedge \mathbf{b}\|_H$ denotes the number of bits that are 0 in $\mathbf{q}$ and 1 in $\mathbf{b}$ (where $\neg$ and $\wedge$ are bitwise negation and logical AND operators). Therefore, all codes with the same value of $r_1(\mathbf{q}, \mathbf{b})$ and $r_2(\mathbf{q}, \mathbf{b})$ lie at the same angle from the query. We say that the code $\mathbf{b}$ *lies at the distance tuple* (e, f) *from* $\mathbf{q}$ or equivalently $\mathbf{b}$ *is a* (e, f)*-neighbor of* $\mathbf{q}$, if we have $(e, f) = (r_1(\mathbf{q}, \mathbf{b}), r_2(\mathbf{q}, \mathbf{b}))$.

As discussed previously, to carry out KNN search in the Hamming space, one can increase the search radius until K neighbors are retrieved. To employ a similar approach for solving the angular KNN, we must first find the correct sequence of tuples that corresponds to the decreasing values of sim and then for each tuple such as (e, f) the HWT tree must be searched to retrieve the binary codes lying at distance tuple (e, f). The correct sequence of tuples can be found efficiently using the sequential algorithm proposed in [45]. We next show how to search the HWT in order to find codes lying at a specific distance tuple.

Note that the binary codes that correspond to the tuple (e, f) lie at the Hamming distance $e + f$ from the query. Therefore, a simple solution for retrieving desired codes involves solving the r-neighbor problem with $r = a + b$ and then linearly scanning retrieved candidates to find the codes corresponding with tuple (e, f). However, this approach may search some unnecessary nodes. For example, suppose we are looking for codes lying at distance tuple (2,2) from the query with $\|\mathbf{q}\|_1 = 64$ in a depth-one Hamming weight tree for 128-bit codes. Our algorithm would search among the nodes with Hamming weights 60,62,64,66,68 to find the codes lying at Hamming distance of 4 from the query. However, it is easy to see that only node 64 must be searched because codes lying at distance tuple (2,2), have equal Hamming weights to that of the query.

To find all (e, f) neighbors of $\mathbf{q}$, we need to determine the nodes that the search algorithm must recurse on. To this aim, we first show the relationship between the Hamming weight patterns of two codes lying at a specific distance tuple.

Proposition 3. *For binary code* $\mathbf{b}$ *lying at distance tuple* (e, f) *from* $\mathbf{q}$*, we have:*

$$\sum_{i=1}^{d}\left[\|\mathbf{q}_d^{(i)}\|_H - \|\mathbf{b}_d^{(i)}\|_H\right]_+ \leq e, \tag{12}$$

$$\sum_{i=1}^{d}\left[\|\mathbf{b}_d^{(i)}\|_H - \|\mathbf{q}_d^{(i)}\|_H\right]_+ \leq f, \tag{13}$$

$$\|\mathbf{q}\|_H + (f - e) = \|\mathbf{b}\|_H, \tag{14}$$

where $[.]_+ = \max(0, .)$ *is the standard hinge loss function.*

Proof. If (12) is not satisfied then at least $e + 1$ set bits in $\mathbf{q}$ are flipped to zero in $\mathbf{b}$ which contradicts the fact that $\mathbf{b}$ lies at distance tuple (e, f) from $\mathbf{q}$. Also, if (13) is not satisfied, at least $f + 1$ clear bits in $\mathbf{q}$ are flipped to one in $\mathbf{b}$. (14) can be easily derived from the fact each distance tuple uniquely specifies the Hamming weights of the codes of interest. □

To find all the points that lie at the distance tuple (e, f) in a Hamming weight tree, we again have two options: i) scan all children and recurse on nodes satisfying conditions (12), (13), (14), ii) compute all possible Hamming weight patterns that satisfy (12), (13), (14) and directly recurse on those nodes.

To use the second option, at a HWT node with Hamming weight pattern of $[\phi_1, \ldots, \phi_w]$, the algorithm finds the promising children by solving the following system of equations and the recurse on the children with the Hamming weight pattern of $[x_1, \ldots, x_{2w}]$.

$$\begin{cases} x_1 + x_2 = \phi_1 \\ \ldots \\ x_{2w-1} + x_{2w} = \phi_w \\ \sum_{i=1}^{2w}\left[\|\mathbf{q}_d^{(i)}\|_H - x_i\right]_+ \leq e \\ \sum_{i=1}^{2w}\left[x_i - \|\mathbf{q}_d^{(i)}\|_H\right]_+ \leq f \\ x_i \in \mathbb{N}_0 \\ \|\mathbf{q}\|_H + (f - e) = \sum_i^{2w} x_i \end{cases}. \tag{15}$$

The last condition in (15) can be easily satisfied by simply forcing the search algorithm to recurse on only one node at the first level which has the pattern of $\|\mathbf{q}\|_H + (f - e)$. Doing that, (15) can be rewritten as:

$$\begin{cases} \sum_{i=1}^{w}\left[\|\mathbf{q}_d^{(2i-1)}\|_H - x_{2i-1}\right]_+ + \left[\|\mathbf{q}_d^{(2i)}\|_H - \phi_i + x_{2i-1}\right]_+ \leq e \\ \sum_{i=1}^{w}\left[x_{2i-1} - \|\mathbf{q}_d^{(2i-1)}\|_H\right]_+ + \left[\phi_i - x_{2i-1} - \|\mathbf{q}_d^{(2i)}\|_H\right]_+ \leq f \\ x_i \in \mathbb{N}_0 \end{cases}. \tag{16}$$

By using the fact that $[a]_+ = \frac{1}{2}(a + |a|)$, we have:

$$\begin{cases} \sum_{i=1}^{w}\left|\|\mathbf{q}_d^{(2i-1)}\|_H - x_{2i-1}\right| + \left|\|\mathbf{q}_d^{(2i)}\|_H - \phi_i + x_{2i-1}\right| \leq \\ \qquad 2e - \sum_{i=1}^{w}\left(\|\mathbf{q}_d^{(2i-1)}\|_H + \|\mathbf{q}_d^{(2i)}\|_H - \phi_i\right) \\ \sum_{i=1}^{w}\left|\|\mathbf{q}_d^{(2i-1)}\|_H - x_{2i-1}\right| + \left|\|\mathbf{q}_d^{(2i)}\|_H - \phi_i + x_{2i-1}\right| \leq \\ \qquad 2f + \sum_{i=1}^{w}\left(\|\mathbf{q}_d^{(2i-1)}\|_H + \|\mathbf{q}_d^{(2i)}\|_H - \phi_i\right) \\ x_i \in \mathbb{N}_0 \end{cases}. \tag{17}$$

Considering that $\sum_{i=1}^{w}(\|\mathbf{q}_d^{(2i-1)}\|_H + \|\mathbf{q}_d^{(2i)}\|_H - \phi_i) = e - f$, solving (15) reduces to finding all solutions of the following inequality:

$$\sum_{i=1}^{w}\left|\|\mathbf{q}_d^{(2i-1)}\|_H - x_{2i-1}\right| + \left|\|\mathbf{q}_d^{(2i)}\|_H - \phi_i + x_{2i-1}\right| \leq e + f, \tag{18}$$

which is same as (6) with $r = e + f$. Thus, the search algorithm recurses on the nodes satisfying two conditions. First, they should satisfy the weight condition, $\|\mathbf{q}\|_H + (f - e) = \sum_{i=1}^{w}\phi_w$. Second, they must be a $(e + f, w)$-neighbor pattern of the query. The first condition is satisfied by making sure that at the first level the algorithm selects the node with weight $\|\mathbf{q}\|_H + (f - e)$. Then, to satisfy the second condition, the algorithm solves the $(e + f)$-near neighbor problem on the subtree of the selected node. Therefore, as in the Hamming KNN, angular KNN search can performed by only solving instances of the r-neighbor problem.

5 Experimental Results

In this section, we empirically gauge the performance of HWT in comparison with the linear scan baseline, MIH of [9], Angular MIH (AMIH) of [45], and four popular tree-based search algorithms namely Annoy, kd tree, ball tree and RP forest. The following experiments are run on a single core 2.0 GHz CPU with 128 GB of RAM. Linear scan and HWT are both coded in C++ and compiled with GCC 4.4.4 using the same flags. We used the publicly available implementation of MIH and AMIH in our experiments.

5.1 Datasets

We evaluate the performance of HWT on two well-known real-world datasets which are publicly available: (1) ANN_1B [52] with 1 billion 128D SIFT vectors, and (2) 80 millions 384D GIST descriptors from the 80 million tiny images [53]. 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 map real-valued SIFT and GIST vectors to binary codes, for the Hamming distance experiments, we use the well-known hyperplane LSH [54] 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 the angular distance, we implemented AQBC [20] and applied on the real-valued vectors to produce angular preserving binary codes. We also make our implementation of AQBC publicly available (`github.com/sepehr3pehr/AQBC`). As opposed to the hyperplane LSH which is a randomized technique with no learning phase, AQBC is data-depedent and requires the parameters of the hash function to be learned. The ANN_1B dataset comes with a predefined *learning* set of 100 million SIFT vectors from which we randomly select one million points. Similarly, for 80M Gist descriptors dataset, we randomly select 300K points from the dataset to form the learning set.

For each dataset and similarity measure, we generate 32, 64 and 128 bit binary datasets. With two datasets, three different code lengths, and two distance measures, we obtain 12 binary datasets. While it is possible to reorder the bits sequence in order to achieve better query time [9], [55], for the sake of simplicity, we do not adopt such optimizations in our experiments.

5.2 Results

5.2.1 *Effect of Threshold Value*

We first investigate the effect of parameter τ 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 τ form shallow trees and therefore less pruning takes place which increases the required number of 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 τ to be sufficiently large. Such a tree exhibits a performance similar to linear scan. On the other hand, small values of τ create trees with higher depths which causes further pruning but more node traversals and higher storage cost.

Fig. 5 shows the average query time for different values of τ. The figure indicates that smaller values of τ result 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 τ. Interestingly, this parameter can be set based on the available memory of the target platform to balance the query time and memory requirement. We observed similar patterns for the angular distance but the results are omitted due to the space limit.

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. However, the dominant trend is that the query time increases with the size of dataset.

For the following experiments, we set $\tau = 1000$. For this choice, 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 [9].

5.2.2 *HWT versus 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 1 reports the average query time of the linear scan baseline and HWT along with speed up factors gained by using HWT for different KNN problems. For a large range of code lengths and different values of K, HWT can achieve orders of magnitude

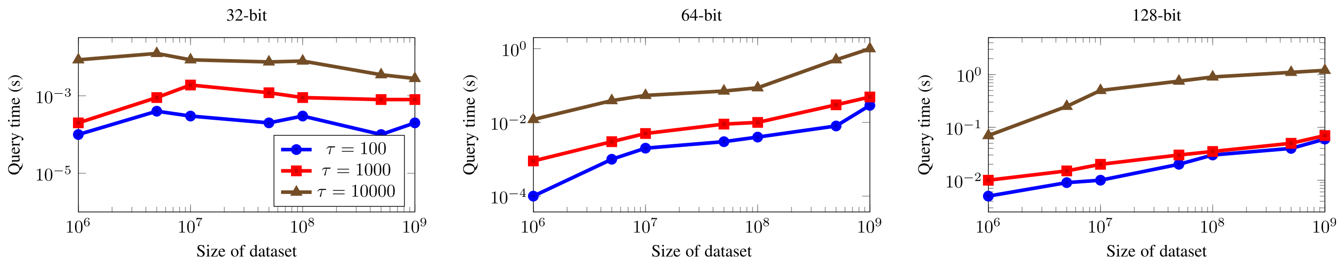


Fig. 5. Average query time of the nearest neighbor search for τ=100, 1000 and 10000 on ANN_1B dataset.

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. Also for longer codes, the difference between HWT and linear scan becomes smaller. Fig. 6 also plots the average query time of HWT and linear scan for different dataset sizes which clearly shows the superiority of HWT.

5.2.3 HWT versus MIH

We also compare the performance of HWT with MIH [9] and angular MIH [45] which to the best of our knowledge have the best query time for solving the exact NN problem for Hamming and angular distances. To set the number of hash tables for MIH, Norouzi et al. [41] used a hold-out validation set of the dataset 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 and AMIH with different values of m for different sizes of the dataset to investigate the relative performance of these two technique. Not surprisingly, there is a natural trade-off between large and small values of m. Too 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. In the extreme case if one bit is assigned to each hash table then we have $m = p$ and the query must be compared with all points. On the other hand, too small values lead to forming hash tables with many empty buckets. In this case, to retrieve K nearest neighbors, the required radius of search per hash table must be increased. This translates to checking many empty buckets.

Figs. 7 and 8 show the average query time of MIH versus HWT for Hamming distance and HWT versus AMIH for the angular distance applied to the task of nearest neighbor search (1NN). For MIH and AMIH, 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 dataset size in which MIH and AMIH 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 and AMIH in their optimal parameter setting have a marginally better performance than HWT 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.

TABLE 1
Average Running Time of Nearest Neighbor Search with HWT and Linear Scan (LS) Algorithms on ANN_1B and GIST 80M

				Hamming		Angular	
	#bits	Method	K	Time	Speedup	Time	Speedup
ANN_1B	32	LS	-	18.14	1×	30.65	1×
		HWT	1	0.0008	22675×	0.0084	3649×
		HWT	10	0.0055	3088×	0.0099	3095×
		HWT	100	0.015	1029×	0.034	901×
	64	LS	-	22.47	1×	31.78	1×
		HWT	1	0.049	458×	0.029	1095×
		HWT	10	0.078	150×	0.084	378×
		HWT	100	0.249	90×	0.39	81×
	128	LS	-	32.11	1×	49.34	1×
		HWT	1	0.07	459×	0.25	197×
		HWT	10	0.09	356×	0.38	129×
		HWT	100	0.366	88×	0.79	62×
GIST 80M	32	LS	-	1.02	1×	3.05	1×
		HWT	1	0.003	340×	0.006	508×
		HWT	10	0.005	204×	0.009	338×
		HWT	100	0.003	340×	0.012	254×
	64	LS	-	1.22	1×	3.79	1×
		HWT	1	0.009	113×	0.019	199×
		HWT	10	0.015	81×	0.037	102×
		HWT	100	0.053	23×	0.077	49×
	128	LS	-	2.5	1×	4.93	1×
		HWT	1	0.08	31×	0.15	32×
		HWT	10	0.15	16×	0.39	12×
		HWT	100	0.5	5×	0.71	7 ×

5.2.4 HWT versus Tree Search Algorithms

Next, we compare the performance of HWT with some of the well-known branch and bound nearest neighbor search techniques. For the experiments of this section we use the *ann-benchmark* [56], a tool for evaluating the performance of in-memory approximate nearest neighbor search. This tool

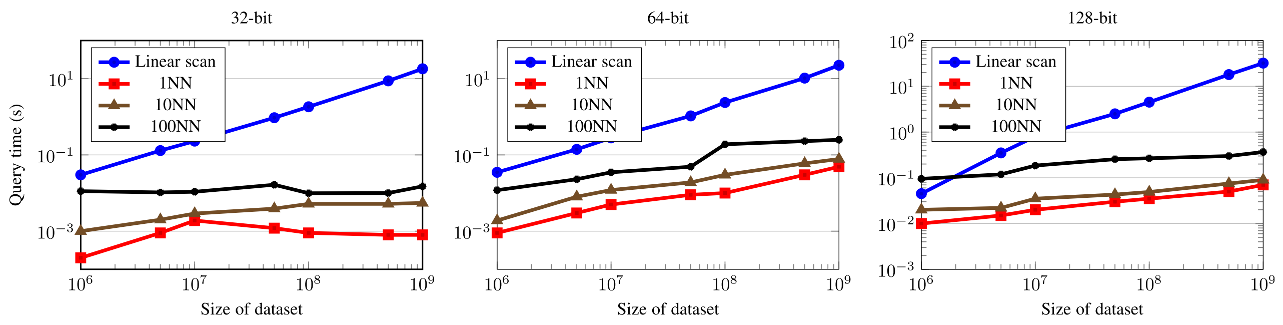


Fig. 6. Average query time of HWT and linear scan on the ANN_1B dataset for solving the Hamming NNS.

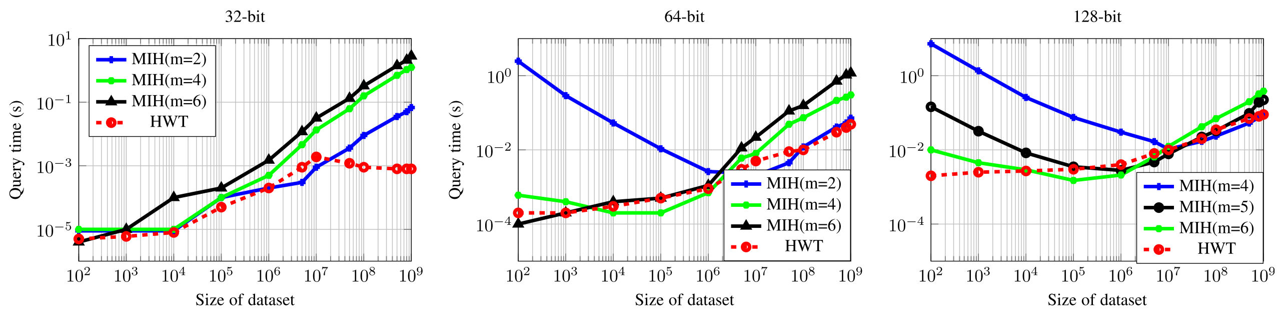


Fig. 7. Average query time of HWT and MIH for the task of Hamming nearest neighbor search. The value of m denotes the number of hash tables used in MIH.

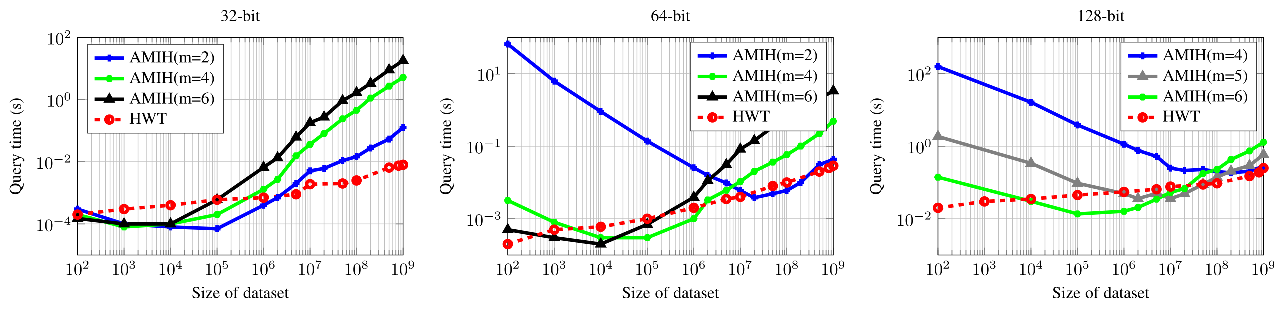


Fig. 8. Average query time of HWT and AMIH for the task of angular nearest neighbor search.

provides a standard interface for measuring the performance and quality achieved by nearest neighbor algorithms on different standard datasets. To make our algorithm usable by ann-benchmark, we implemented a python wrapper for our code and added the parameters to be tested to the configuration file. We compare HWT with tree-based search algorithms of ann-benchmark that support Hamming distance, namely Annoy, KD tree, Ball Tree and RP forest. Discussing the details of these techniques is beyond the scope of this paper but we briefly introduce them here.

Annoy [57] decomposes the search space using multiple trees to achieve sublinear search time. Each intermediate node splits the space into two half-spaces by sampling two points from the subset and taking the hyperplane equidistant from the them. Each leaf node stores a subset of data points that lie in the region of space defined by its ancestors. Given the indexed dataset, the search algorithm prunes the search by only considering the points in the subspace where the query falls. Annoy incorporates a forest of such trees to increase the chance of colliding query with the nearest neighbor in at least of the leaves.

KD tree [42] is one of the long-standing nearest neighbor search algorithms. Each intermediate node in KD tree selects one of the input dimensions as the discriminators to partition the space using axis-aligned hyper-planes. Therefore, each leaf node represents a subset of point lying in the hypercube described by its ancestors.

Ball tree [58], as the name suggests, partitions the input space into disjoint hyperspheres each represented with a center and a diameter. While hyperspheres may intersect, each point is assigned to one ball per level according to its distance from the centers.

RP forest [59] works by creating a set of binary random projection trees. In each tree, dataset points are recursively partitioned based on the cosine of the angle of the points and a randomly drawn hyper-plane where the median angle is used as the pivot point.

Each of the selected algorithms are called with a set of parameters which directly affect the query-time versus recall rate trade-off. We run all algorithms multiple times each with a different parameter setting using the same configurations suggested by the ann-benchmark. To have a fair comparison with our exact technique, among the different settings we report the results of the one with the highest recall rate, defined as ratio of the number of points in the retrieved items being the true nearest neighbors and the

TABLE 2
Performance of Different Tree Based Techniques Applied to the 256-bit SIFT Dataset with 1 Million Items to Find the Hamming Nearest Neighbor

Method	Time (ms)	Memory (KB)	Recall	Indexing (min)
Annoy	9.32	24169	99.23	32
Ball tree	197.32	1224	1	24
KD tree	181.30	1486	98.12	78
RP forest	34.12	9486	97.23	37
HWT	4.36	14896	1	24

number K of the true nearest neighbors. The dataset used for the experiments is the 1 million 256-bit SIFT dataset provided along with the ann-benchmark.

Table 2 compares the performance of HWT against the four selected tree based algorithms. Results indicate that HWT significantly outperforms traditional tree indexing approaches KD tree and Ball tree. The proposed approach also achieves a modest speed-up of $2\times$ speed-up compared to the state-of-the-art Annoy with less memory overhead. For the angular distance, among the four selected techniques only Annoy supports angular nearest neighbor search which achieves average query time of 0.091 seconds whereas HWT takes 0.064 seconds. It is worth mentioning that the tree-based algorithms used in this section are more general techniques capable of working with several distance measures in addition to the Hamming distance. We believe that modifying them to better handle binary data through bitwise manipulations can result in performance boost but doing so requires a considerable amount of human effort and is out of the scope of this research.

6 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 exhibit superior performance than MIH and AMIH for dynamic applications. 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 specially when the size of the dataset is large.

While the basic algorithm is developed in this study, we conjecture that similar ideas can be used for other metrics domains by partitioning the search space based on the norms of the vectors. Also, it is interesting to investigate approximate solutions based on the same Hamming weight tree data structure. For example, one can stop the search procedure once a fixed number of unique candidates are retrieved.

Acknowledgments

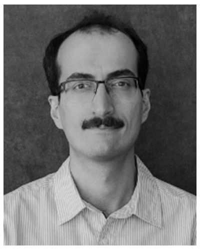
This research was supported by Natural Sciences and Engineering Research Council (NSERC) Canada, Ontario Research Fund: Research Excellence (ORF-RE) and Collaborative Research and Development (CRD). The authors would like to express their appreciations to the associate editor and the anonymous reviewers for their valuable suggestions.

References

[1] M. Raginsky and S. Lazebnik, "Locality-sensitive binary codes from shift-invariant kernels," in *Proc. Adv. Neural Inf. Process. Syst.*, 2009, pp. 1509–1517.
[2] A. Torralba, R. Fergus, and Y. Weiss, "Small codes and large image databases for recognition," in *Proc. IEEE Conf. Comput. Vis. Pattern Recognit.*, 2008, pp. 1–8.
[3] H. Jegou, M. Douze, and C. Schmid, "Hamming embedding and weak geometric consistency for large scale image search," in *Proc. Eur. Conf. Comput. Vis.*, 2008, pp. 304–317.
[4] D. Kuettel, M. Guillaumin, and V. Ferrari, "Segmentation propagation in imagenet," in *Proc. Eur. Conf. Comput. Vis.*, 2012, pp. 459–473.
[5] G. Shakhnarovich, P. Viola, and T. Darrell, "Fast pose estimation with parameter-sensitive hashing," in *Proc. IEEE Conf. Comput. Vis.*, 2003, Art. no. 750.
[6] P. Bühlmann, P. Drineas, M. Kane, and M. van der Laan, *Handbook of Big Data*. Boca Raton, FL, USA: CRC Press, 2016.
[7] M. M. Esmaeili, R. K. Ward, and M. Fatourechi, "A fast approximate nearest neighbor search algorithm in the hamming space," *IEEE Trans. Pattern Anal. Mach. Intell.*, vol. 34, no. 12, pp. 2481–2488, 2012.
[8] Z. Jiang, L. Xie, X. Deng, W. Xu, and J. Wang, "Fast nearest neighbor search in the hamming space," in *Proc. Int. Conf. Multimedia Model.*, 2016, pp. 325–336.
[9] M. Norouzi, A. Punjani, and D. J. Fleet, "Fast exact search in hamming space with multi-index hashing," *IEEE Trans. Pattern Anal. Mach. Intell.*, vol. 36, no. 6, pp. 1107–1119, 2014.
[10] X. Zhang, J. Qin, W. Wang, Y. Sun, and J. Lu, "Hmsearch: An efficient hamming distance query processing algorithm," in *Proc. Int. Conf. Sci. Statist. Database Manage.*, 2013, Art. no. 19.
[11] O. Chapelle, P. Haffner, and V. N. Vapnik, "Support vector machines for histogram-based image classification," *IEEE Trans. Neural Netw.*, vol. 10, no. 5, pp. 1055–1064, 1999.
[12] M. Hein and O. Bousquet, "Hilbertian metrics and positive definite kernels on probability measures," in *Proc. Int. Conf. Artif. Intell. Statist.*, 2005, pp. 136–143.
[13] M. Calonder, V. Lepetit, C. Strecha, and P. Fua, "Brief: Binary robust independent elementary features," in *Proc. Eur. Conf. Comput. Vis.*, 2010, pp. 778–792.
[14] E. Rublee, V. Rabaud, K. Konolige, and G. Bradski, "Orb: An efficient alternative to sift or surf," in *Proc. IEEE Conf. Comput. Vis.*, 2011, pp. 2564–2571.
[15] S. Leutenegger, M. Chli, and R. Y. Siegwart, "Brisk: Binary robust invariant scalable keypoints," in *Proc. IEEE Conf. Comput. Vis.*, 2011, pp. 2548–2555.
[16] D. G. Lowe, "Distinctive image features from scale-invariant keypoints," *Int. J. Comput. Vis.*, vol. 60, no. 2, pp. 91–110, 2004.
[17] R. Salakhutdinov and G. Hinton, "Semantic hashing," *Int. J. Approximate Reasoning*, vol. 50, no. 7, pp. 969–978, 2009.
[18] C. Da, S. Xu, K. Ding, G. Meng, S. Xiang, and C. Pan, "Amvh: Asymmetric multi-valued hashing," in *Proc. IEEE Conf. Comput. Vis. Pattern Recognit.*, 2017, pp. 898–906.
[19] Y. Gong, S. Lazebnik, A. Gordo, and F. Perronnin, "Iterative quantization: A procrustean approach to learning binary codes for large-scale image retrieval," *IEEE Trans. Pattern Anal. Mach. Intell.*, vol. 35, no. 12, pp. 2916–2929, 2013.
[20] Y. Gong, S. Kumar, V. Verma, and S. Lazebnik, "Angular quantization-based binary codes for fast similarity search," in *Proc. Adv. Neural Inf. Process. Syst.*, 2012, pp. 1196–1204.
[21] B. Kulis and K. Grauman, "Kernelized locality-sensitive hashing for scalable image search," in *Proc. IEEE Conf. Comput. Vis.*, 2009, pp. 2130–2137.
[22] K. Lin, J. Lu, C. S. Chen, J. Zhou, and M. T. Sun, "Unsupervised deep learning of compact binary descriptors," *IEEE Trans. Pattern Anal. Mach. Intell.*, 2018, pp. 1–1.
[23] L. Liu, L. Shao, F. Shen, and M. Yu, "Discretely coding semantic rank orders for supervised image hashing," in *Proc. IEEE Conf. Comput. Vis. Pattern Recognit.*, 2017, pp. 5140–5149.
[24] M. Norouzi and D. M. Blei, "Minimal loss hashing for compact binary codes," in *Proc. Int. Conf. Mach. Learn.*, 2011, pp. 353–360.
[25] R. Raziperchikolaei and M. A. Carreira-Perpinán, "Learning independent, diverse binary hash functions: Pruning and locality," in *Proc. Int. Conf. Data Mining*, 2016, pp. 1173–1178.

[26] F. Shen, C. Shen, W. Liu, and H. T. Shen, “Supervised discrete hashing,” in *Proc. IEEE Conf. Comput. Vis. Pattern Recognit.*, 2015, pp. 37–45.
[27] Y. Weiss, A. Torralba, and R. Fergus, “Spectral hashing,” in *Proc. Adv. Neural Inf. Process. Syst.*, 2009, pp. 1753–1760.
[28] E.-J. Ong and M. Bober, “Improved hamming distance search using variable length substrings,” in *Proc. IEEE Conf. Comput. Vis. Pattern Recognit.*, 2016, pp. 2000–2008.
[29] V. Balntas, L. Tang, and K. Mikolajczyk, “Binary online learned descriptors,” *IEEE Trans. Pattern Anal. Mach. Intell.*, vol. 40, no. 3, pp. 555–567, Mar. 2018.
[30] L. K. Huang, Q. Yang, and W. S. Zheng, “Online hashing,” *IEEE Trans. Neural Netw. Learn. Syst.*, vol. 29, no. 6, pp. 2309–2322, Jun. 2017.
[31] M. Minsky and S. Papert, *Perceptrons.* Cambridge, MA, USA: MIT Press, 1969.
[32] H. Liu, R. Wang, S. Shan, and X. Chen, “Deep supervised hashing for fast image retrieval,” in *Proc. IEEE Conf. Comput. Vis. Pattern Recognit.*, Jun. 2016, pp. 2064–2072.
[33] Y. Ke, R. Sukthankar, L. Huston, Y. Ke, and R. Sukthankar, “Efficient near-duplicate detection and sub-image retrieval,” *ACM Multimedia*, vol. 4, 2004, Art. no. 5.
[34] O. Barkol and Y. Rabani, “Tighter bounds for nearest neighbor search and related problems in the cell probe model,” in *Proc. Annu. ACM Symp. Theory Comput.*, 2000, pp. 388–396.
[35] J. Alman and R. Williams, “Probabilistic polynomials and hamming nearest neighbors,” in *Proc. Annu. Symp. Found. Comput. Sci.*, 2015, pp. 136–150.
[36] P. Indyk and R. Motwani, “Approximate nearest neighbors: towards removing the curse of dimensionality,” in *Proc. Annu. ACM Symp. Theory Comput.*, 1998, pp. 604–613.
[37] W. B. Johnson and J. Lindenstrauss, “Extensions of lipschitz mappings into a hilbert space,” *Contemporary Math.*, vol. 26, no. 189–206, 1984, Art. no. 1.
[38] M. Pătraşcu, “Lower bound techniques for data structures,” Ph.D. dissertation, Dept. Electr. Eng. Comput. Sci., Massachusetts Inst. Technol., Cambridge, MA, USA, 2008.
[39] A. Beygelzimer, S. Kakade, and J. Langford, “Cover trees for nearest neighbor,” in *Proc. ACM Int. Conf. Mach. Learn.*, 2006, pp. 97–104.
[40] M. Muja and D. G. Lowe, “Scalable nearest neighbor algorithms for high dimensional data,” *IEEE Trans. Pattern Anal. Mach. Intell.*, vol. 36, no. 11, pp. 2227–2240, 2014.
[41] M. Norouzi, A. Punjani, and D. J. Fleet, “Fast search in hamming space with multi-index hashing,” in *Proc. IEEE Conf. Comput. Vis. Pattern Recognit.*, 2012, pp. 3108–3115.
[42] J. L. Bentley, “Multidimensional binary search trees used for associative searching,” *Commun. ACM*, vol. 18, no. 9, pp. 509–517, 1975.
[43] F. Aurenhammer, “Voronoi diagrams—A survey of a fundamental geometric data structure,” *ACM Comput. Surveys*, vol. 23, no. 3, pp. 345–405, 1991.
[44] H. Samet, *Foundations of Multidimensional and Metric Data Structures.* San Mateo, CA, USA: Morgan Kaufmann, 2006.
[45] S. Eghbali and L. Tahvildari, “Fast cosine similarity search in binary space with angular multi-index hashing,” *IEEE Trans. Knowl. Data Eng.*, vol. 31, no. 2, pp. 329–342, Feb. 2019.
[46] D. Greene, M. Parnas, and F. Yao, “Multi-index hashing for information retrieval,” in *Proc. Annu. Symp. Found. Comput. Sci.*, 1994, pp. 722–731.
[47] S. Eghbali, H. Ashtiani, and L. Tahvildari, “Online nearest neighbor search in binary space,” in *Proc. Int. Conf. Data Mining*, Nov. 2017, pp. 853–858.
[48] M. Muja and D. G. Lowe, “Fast matching of binary features,” in *Proc. Conf. Comput. Robot Vis.*, 2012, pp. 404–410.
[49] J. Gao, H. Jagadish, B. C. Ooi, and S. Wang, “Selective hashing: Closing the gap between radius search and k-nn search,” in *Proc. Int. Conf. Knowl. Discovery Data Mining*, 2015, pp. 349–358.
[50] R. J. Bayardo, Y. Ma, and R. Srikant, “Scaling up all pairs similarity search,” in *Proc. Int. Conf. World Wide Web*, 2007, pp. 131–140.
[51] A. Shrivastava and P. Li, “In defense of minhash over simhash,” in *Proc. Artif. Intell. Statist.*, 2014, pp. 886–894.
[52] H. Jégou, R. Tavenard, M. Douze, and L. Amsaleg, “Searching in one billion vectors: Re-rank with source coding,” in *Proc. Int. Conf. Acoust., Speech Signal Process.*, 2011, pp. 861–864.
[53] A. Torralba, R. Fergus, and W. T. Freeman, “80 million tiny images: A large data set for nonparametric object and scene recognition,” *IEEE Trans. Pattern Anal. Mach. Intell.*, vol. 30, no. 11, pp. 1958–1970, Nov. 2008.
[54] M. S. Charikar, “Similarity estimation techniques from rounding algorithms,” in *Proc. ACM Symp. Theory Comput.*, 2002, pp. 380–388.
[55] V. Mic, D. Novak, and P. Zezula, “Modifying hamming spaces for efficient search,” in *Proc. Int. Conf. Data Mining Workshops*, 2018, pp. 945–953.
[56] M. Aumüller, E. Bernhardsson, and A. Faithfull, “Ann-benchmarks: A benchmarking tool for approximate nearest neighbor algorithms,” *Proc. Int. Conf. Similarity Search Appl.*, 2017, pp. 34–49.
[57] E. Bernhardsson, “Annoy github.com/spotify/annoy.’’
[58] S. M. Omohundro, “Five Balltree Construction Algorithms,” International Computer Science Institute, 1989.
[59] “Rpforrest github.com/lyst/rpforest.”

Sepehr Eghbali received the BSc degree in computer engineering from the Isfahan University of Technology, Iran, in 2008 and the MSc degree in artificial intelligence and robotics from the University of Tehran, in 2012. He is working toward the PhD degree in the Department of Electrical and Computer Engineering, University of Waterloo. His current research interests include fast retrieval on massive and high-dimensional datasets.


Hassan Ashtiani received the bachelor’s degree in computer engineering and the master’s degree in AI and Robotics, both from University of Tehran, and the PhD degree in computer science from the University of Waterloo, in 2018. He is an assistant professor with the Department of Computing and Software, McMaster University. He is interested in a broad range of problems in the intersection of theoretical and applied machine learning. In the recent years, he has particularly focused on building and analyzing new models for unsupervised and semi-supervised learning.

Ladan Tahvildari is a professor with the Department of Electrical and Computer Engineering, University of Waterloo. She founded the Software Technologies Applied Research Laboratory at the University of Waterloo in 2004. Since then, she has led numerous research and innovation activities in the area of dynamic software evolution, with emphasis on self-adaptive systems and testing approaches. Together with her students and collaborators, she works on the design and development of novel models, architectures, and decision-making approaches to improve the quality of software systems. She has published widely on these topics and collaborates extensively with high-tech companies and non-profit organizations to ensure real-world applicability of her research contributions. She is a member of the Association for Computing Machinery (ACM), ACM Special Interest Group on Software Engineering (SIGSOFT), and the Institute of Electrical and Electronics Engineers (IEEE). She also has more than 10 years of experience as chair of the IEEE Women in Engineering (WIE) Affinity Group in K-W Section.air of the IEEE Computer Society (CS), and IEEE Women in Engineering Affinity Group in the local chapter since 2004. Various awards have recognized her research accomplishments. Recently, she has been honoured with the prestigious Ontario’s Early Researcher Award (ERA) to recognize her work in self-adaptive software.