An Index Structure for Fast Range Search in Hamming Space

by

Ernesto Rodriguez Reina

A Thesis Submitted in Partial Fulfillment
of the Requirements for the Degree of

Master in Computer Science

in

Faculty of Science
Computer Science

University of Ontario Institute of Technology

November 2014

Abstract

An Index Structure for Fast Range Search in Hamming Space

Ernesto Rodriguez Reina

Master in Computer Science

Faculty of Science

University of Ontario Institute of Technology

2014

This thesis addresses the problem of indexing and querying very large databases of binary vectors. Such databases of binary vectors are a common occurrence in domains such as information retrieval and computer vision. We propose an indexing structure consisting of a compressed trie and a hash table for supporting range queries in Hamming space. The index structure, which can be updated incrementally, is able to solve the range queries for any radius. Out approach minimizes the number of memory access, and as result significantly outperforms state-of-the-art approaches.

Keywords: range queries, r-neighbors queries, hamming distance.

To my beloved wife

for being my source of inspiration.

Acknowledgements

I would like to express my special appreciation and thanks to my advisors Professor Dr. Ken Pu and Professor Dr. Faisal Qureshi for the useful comments, remarks and engagement throughout the learning process of this master thesis. You both have been tremendous mentors for me. Your advices on both research as well as on my career have been priceless.

I owe my deepest gratitude to my college Luis Zarrabeitia for introducing me to Professor Qureshi and Professor Pu, and also for all your help to came to study to the UOIT.

A special thanks to my mom. Your prayer for me was what sustained me thus far.

Last but not the least, I would like to express my deeply thanks my lovely wife, who have supported me throughout entire process, both by keeping me harmonious and helping me putting pieces together. I will be grateful forever for your love.

Contents

List of Figures

List of Algorithms

Chapter 1

Introduction

Increasingly many applications in domains ranging from image matching to information retrieval generate and analyze very large number of multidimensional feature descriptors. These feature descriptors are often encoded as binary vectors because binary vectors can be efficiently stored and indexed. Furthermore, many schemes exist to search a binary vector in a given collection. To search over a set of binary vectors it is necessary to define a distance (or dissimilarity) metric over binary vector space. A common technique for computing "distance" between two binary vectors (of equal length) is to use *Hamming distance*. The Hamming distance between two vectors is equal to the the number of positions in which they differ.

Given a set of binary vectors D and a query vector $\mathbf{q}$, finding exact matches is often not sufficient. Rather most applications seek to find the subset of D that are within some distance (say, Hamming distance) of the query vector $\mathbf{q}$ [32]. We refer to the problem of finding vectors in D that are within some Hamming distance of $\mathbf{q}$ as the *r-neighbors search problem*. Another way to state this is that we want to find all r-neighbors (present in set

D) of the given query vector $\mathbf{q}$. Here, we define that a binary vector is an r-neighbour of a query vector $\mathbf{q}$ if it differs from $\mathbf{q}$ in r bits or less. In some domains, this problem is known as *Approximate Query* problem [18], the *Point Location in Equal Balls* (PLEB) problem [20], and *Hamming Distance Range query* problem [26]. The r-neighbour search problem arises in many different applications, such as, image search [24, 23, 39, 38], image classification [5], object segmentation [22], parameter estimation [36], chemicals search [15], audio and video content retrieval & preference matching [40, 29, 33, 12], iris matching [41], web-pages duplication detection [28, 37], to name a few.

Here's a concrete example of how r-neighbors search is used for image matching, search, and retrieval [24]. First, each image in the collection is encoded as a set of local binary descriptors [34, 8, 1]. Next, binary descriptors collected over the entire collection are indexed into a data structure that supports fast r-neighbors queries or k-nearest-neighbors queries. An inverted index matches each stored vectors (descriptors) to the image that contains it. Binary descriptors computed from the query image are compared against the database to find the set of "closest" vectors in the database. Each of these vectors point to an image in the collection and voting is performed to identify the image (stored in the collection) that best matches the query image. Details can be found in Landré & Truchetet 2007 [24].

The above example outlines a common strategy used for image retrieval using binary descriptors. The key challenge here is one of scale. For example, say we encode each image using 1,000 128-bit binary vectors. This suggests that even a moderate size collection of one million images will have 1,000,000,000 binary vectors. At query time, for each image, a naïve approache would require 1,000,000,000 $\times$ 1,000 comparisons. Clearly, we need

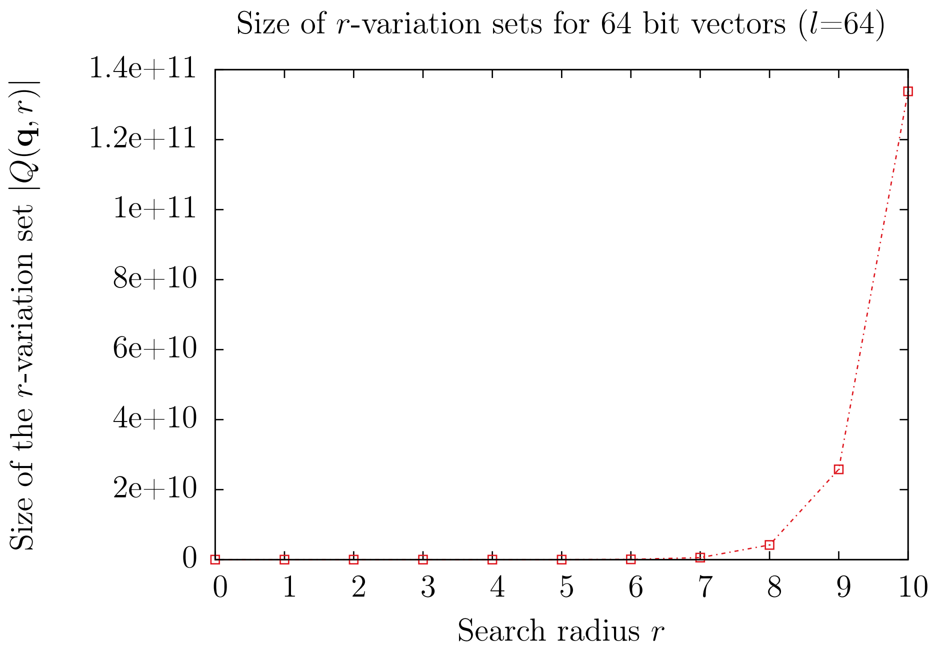

Figure 1.1: Size of the r-variations set for 64 bits vectors. Notice the size grows exponentially with increasing values for r.

efficient methods to deal with this problem. The work presented here is a step in that direction.

1.1 r-neighbors search using hashing

Hamming distance can be computed efficiently via the *xor* operation followed by a *bit count*; however, a linear scan could be prohibitively expensive for large databases. Binary vectors can be used as direct indices (addresses) into a hash table yielding a dramatic increase in search speed over that of an exhaustive linear scan [38]. When addressing the r-neighbors search problem using a hash table populated with the binary vectors in the database D, the naïve approach examines every hash bucket whose indices are within r bits of a query $\mathbf{q}$ (e.g., [38]). To examine these buckets, a set of queries $Q(\mathbf{q}, r)$

is generated and for each vector in that set a hash table lookup is performed. The set $Q(\mathbf{q}, r)$ is called the r-variation of $\mathbf{q}$. For binary vectors of length l, the size of the r-variation set is:

$$|Q(\mathbf{q}, r)| = L(l, r) = \sum_{z=0}^{r} \binom{l}{z}, \tag{1.1}$$

where $r \in [0, l]$. The number of r-variations grows rapidly with the size of the vector l and the search radius r. Consequently, the naïve approach is only practical when dealing with small l and r. For large l or r, it is simply infeasible to examine $|Q(\mathbf{q}, r)|$ buckets. Figure 1.1 plots the size of the r-variations set for 64-bit vectors for different values of r. Notice it grows exponentially with increasing r.

Also, for large l, most of the buckets examined during the above process will be empty. Say a database has n vectors then in general $2^l \gg n$, suggesting that most of the buckets will be empty. For example, the number of r-variations for $l = 64$ and $r = 10$ is more than 133 trillions (1.33×10^{11}). If there are 1 million different vectors in the hash table, the maximum number of non-empty buckets is 1 million (assuming no collisions). That implies that the great majority of lookups performed using the näive approach will be go to empty buckets. We will call those r-variations that end up at an empty bucket as *null r-variations*. Null r-variations lookups are wasted effort and adversely affect the performance of r-neighbour searches. A null r-variation points to an empty bucket and provides no candidates for our query. This thesis presents a novel compressed-trie based data structure that reduces the number of null r-variations lookups and thereby improving the performance of r-neighbors search algorithms.

1.2 r-neighbors search using trie

The curse of r-variation explosion is due to the fact that hash tables, as a data structure, do not support efficient local search. The remedy is to explore other data structures for indexing binary vectors. Say D is a set of Binary vectors and $I(D)$ is the indexing data structure that represents D then an "ideal" indexing structure will have following properties:

1. *lookup efficiency*: $I(D)$ can efficiently answer a membership query: $\mathbf{q} \in D$, given some query vector $\mathbf{q}$; and

2. *local searchability*: Two vectors $\mathbf{d}, \mathbf{d}' \in D$ should be indexed *closely* in $I(D)$ if they are close in Hamming space. This means that a search algorithm can locate both $\mathbf{d}$ and $\mathbf{d}'$ with minimal effort.

Hash tables excel at *lookup efficiency*; however, they fail at *local searchability.* A *trie* (to be described in detail in Section 3.2) is an indexing data structure that provides a more balanced performance characteristics. A trie organizes the binary vectors in $\mathbf{D}$ into a hierarchy (or a tree) based on their prefixes, so that if two binary vectors $\mathbf{d}$ and $\mathbf{d}'$ share a common prefix, then they will be closely positioned in the hierarchy, thus allowing a search algorithm to quickly locate $\mathbf{d}'$ from the location of $\mathbf{d}$ and vice verse.

A trie based approach to solve the r-neighbour problem is to locate the query $\mathbf{q}$, allowing local search in the neighbourhood of $\mathbf{q}$. While a trie is great at satisfying *local searchability*, it is inferior to the hash table with respect to *lookup efficiency.* The motivation of our work is to harness the power of hash tables (thus providing lookup efficiency) and tries (thus providing local searchability) to create a more efficient index for processing r-neighbors queries.

1.3 A Hybrid Approach to the r-Neighbors Search Problem

We are interested in hybrid indexing structures that can merge the hash table and the trie to index a set of binary vectors D. To this end, we propose to index D using both Hash(D) and Trie(D). Rather than processing r-neighbors queries using only one of the two data structures, we propose a query processing algorithm that switches between the two indexing structures back and forth.

The index Trie(D) permits us to use local-search to find a small, but sufficient r-variations, and the much reduced set of r-variations will generate a small number of lookup queries for Hash(D). Thus, the overall algorithm achieves the following:

- the trie only provides r-variation pruning using local-search and does *not* need to resolve full lookup queries; and
- the hash table only needs to handle a small number of lookup queries thanks to the pruning power of a trie index.

1.4 Contributions

In this thesis, we study the problem of r-neighbors searches and propose two techniques for addressing this problem. First, we present a compressed-trie based approach for r-

neighbors searches. We noticed that this approach was unable to achieve the desired performance. After studying the shortcomings of this method, we developed a hybrid compressed-trie plus hash table based approach for r-neighbors searches. We are able to achieve state-of-the-art results using our hybrid approach.

The scientific and algorithmic contribution of this thesis is: a new hybrid compressed-trie + hash table data structure, and the associated query processing algorithm, for indexing binary vectors to support fast r-neighbors searches in Hamming space. The thesis also makes system development and engineering contributions, details of which are listed in the appendix.

1.5 Outline

The remainder of this thesis is organized as follows. We review related work in the next chapter. There we focus on existing schemes that deal with r-neighbors and k nearest neighbors searches using hashing, trie data structures, and hierarchical decomposition methods. The related work, we hope, does an adequate job of juxtaposing our work with the existing techniques. We will also discuss mathematical preliminaries in this chapter. Chapter 3 develops the proposed algorithms. Results are presented in the following chapter. We compare our method with a recent scheme [32] and demonstrate that our method achieves state-of-the-art results. Chapter 5 concludes the thesis with a brief discussion about the strengths and limitations of our method. Relevant technical and implementation details are provided in the appendix.

Chapter 2

Background and Related Work

Here we present the technical background necessary to understand this work. We also summarize existing literature that deals with r-neighbors and nearest neighbors queries on collections of binary vectors.

2.1 Background

2.1.1 Binary Vectors

We begin by defining binary vectors and common operations on these vectors.

Definition 1. *A* binary vector *(also known as Boolean vector)* $\mathbf{v}$ *is a sequence of 0's and 1's. The length of* $\mathbf{v}$ *is denoted as* $|\mathbf{v}|$*, and the element at i-th position* $(0 \leq i < |\mathbf{v}|)$ *is denoted as* $\mathbf{v}[i]$*. The set of all binary vectors of length l is denoted as* 2^l*. The special case of the binary vector of length* 0 *is named* empty vector *and we will denote it as* ϵ*.*

Definition 2. *Given* $\mathbf{x} \in 2^n$ *and* $\mathbf{y} \in 2^m$*. The concatenation of the vectors* $\mathbf{x}$ *and* $\mathbf{y}$

denoted $\mathbf{x}\|\mathbf{y}$ *is the binary vector* $\mathbf{z}$ *where:*

i) $|\mathbf{z}| = n + m$

ii) $\forall i \in \mathbb{Z}^+, i < n : \mathbf{z}[i] = \mathbf{x}[i]$

iii) $\forall i \in \mathbb{Z}^+, i \geq n \wedge i < n + m : \mathbf{z}[i] = \mathbf{y}[i - n]$

Note that the concatenation of any vector $\mathbf{x}$ *with an empty vector is equal to* $\mathbf{x}$*.*

Example 2.1.1. *If* $\mathbf{x} = 10$ *and* $\mathbf{y} = 11$ *then* $\mathbf{z} = \mathbf{x}\|\mathbf{y} = 10\|11 = 1011$*.*

Definition 3. *Given* $\mathbf{x} \in 2^n$ *and* $\mathbf{z} \in 2^m$*. The binary vector* $\mathbf{x}$ *is called* prefix *of* $\mathbf{z}$ *if:*

i) $n \leq m$

ii) $\exists \mathbf{y} \in 2^{m-n} : \mathbf{x}\|\mathbf{y} = \mathbf{z}$

Given a vector $\mathbf{x}$*, its prefix of length* i *is denoted as* $\mathbf{x}[: i]$*. Note that* $\mathbf{x}[: 0] = \epsilon$*. The empty vector is prefix of any vector.*

Example 2.1.2. *If* $\mathbf{z} = 1011$*, then the vectors* $1, 10, 101$ *and* 1011 *are all prefixes of* $\mathbf{z}$*.*

Definition 4. *Given* $\mathbf{x} \in 2^n$ *and* $\mathbf{z} \in 2^m$*. The binary vector* $\mathbf{x}$ *is called* suffix *of* $\mathbf{z}$ *if:*

i) $n \leq m$

ii) $\exists \mathbf{y} \in 2^{m-n} : \mathbf{y}\|\mathbf{x} = \mathbf{z}$

Given a vector $\mathbf{x}$*, its suffix of length* $|\mathbf{x}| - i$ *is denoted as* $\mathbf{x}[i :]$*. Note that* $\mathbf{x}[|\mathbf{x}| :] = \epsilon$*. The empty vector is suffix of any vector.*

Example 2.1.3. *If* $\mathbf{z} = 1011$*, then the vectors* $1, 11, 011$ *and* 1011 *are all suffixes of* $\mathbf{z}$*.*

Definition 5. *Given $D \subseteq 2^m$ and $\mathbf{x} \in 2^n$, $n \leq m$. The binary vector $\mathbf{x}$ is a* prefix of the set of binary vectors D *if* $\forall \mathbf{z} \in D, \exists \mathbf{y} \in 2^{n-m} : \mathbf{x} \| \mathbf{y} = \mathbf{z}$

Example 2.1.4. *If $D = 1011, 1010, 1000$, then the vectors 1 and 10 are all prefixes of the set D.*

Definition 6. *Given $D \subseteq 2^m$ and $\mathbf{x} \in 2^n$. The vector $\mathbf{x}$ is called the* maximum prefix of the set of binary vectors D *if* $\mathbf{x}$ *is a prefix of a set of binary vectors D, and $\neg\exists \mathbf{z} \in 2^q, q > n$ where $\mathbf{z}$ is also prefix of the set of binary vectors D. We denote the maximum common prefix of the set D as $p(D)$.*

Example 2.1.5. *If $D = 1011, 1010, 1000$, then $p(D) = 10$. Note that the vector 1 is also prefix of the set D, but the vector 10 is the largest prefix of that set.*

2.1.2 Hamming Distance and Range Queries

Hamming distance between two binary vectors of equal length is the number of positions at which they differs. Formally: .

Definition 7. *Let $x \oplus y = 0$ if $x = y$ or 1 otherwise ($\oplus$ is the* xor *operator). The Hamming distance between two vectors $\mathbf{x}, \mathbf{y} \in 2^l$ is defined as:*

$$H(\mathbf{x}, \mathbf{y}) = \sum_{i=0}^{l-1} (\mathbf{x}[i] \oplus \mathbf{y}[i])$$

Example 2.1.6. *If $\mathbf{s} = 1011$ and $\mathbf{t} = 1000$ then $H(\mathbf{s}, \mathbf{t}) = 2$. Note that the vectors differs in the last two bits.*

The r-neighbors search problem is defined as finding all vectors in D at Hamming distance r or less of a query vector $\mathbf{q}$:

Definition 8. *The* r-neighbors search *over a set of binary vectors $D \subseteq 2^l$ given the query binary vector $\mathbf{q} \in 2^l$ and a radius $r \in \mathbb{Z}^+$ is defined as finding all vectors in D at Hamming distance r of $\mathbf{q}$:*

$$N_r(D, \mathbf{q}) = \{\mathbf{d} \in D : H(\mathbf{d}, \mathbf{q}) \leq r\}$$

Example 2.1.7. *If $D = 1011, 1010, 1001$, for $\mathbf{q} = 0001$ and $r = 2$ the result set would be the vector* 1001 *that is at Hamming distance 1 and the vector* 1011 *that is at Hamming distance 2.*

There are two variants of r-neighbors search problem: when r is known in advance (all queries use the same value of r) and when r is part of the input. The first case is known as the *static problem* and the second case is known as the *dynamic problem* [26]. This thesis focuses on the solution of the dynamic problem.

2.2 Summary of Existing Techniques

2.2.1 Trie-Based Approaches

There have been several approaches that use the *trie* [16] data structure for the r-neighbor search in Hamming space. The first algorithm that used a trie was proposed by Brodal and Gasieniec [7], it only supported $r = 1$. This algorithm had $O(l)$ query time complexity (l refers to the length of the vector) and $O(ln)$ space complexity (n is the number of vectors in the dataset).

For larger values of r, Arslan and Egecioglu [4] proposed a trie-based algorithm with query time complexity $O(l^{r+2})$. This method was later improved by Arslan reducing the query time complexity to $O(l(\log_{\frac{4}{3}} n - 1)^r(\log_2 n^{r+1}))$ [2]. MaaB and Nowak[27] reduced the query time complexity to $O(l)$ but with space complexity $O(n \log^r n)$.

The fundamental limitation of these methods is that they require that binary vectors fit in a machine word[1], and these methods cannot be used for long vectors. The same author extend this trie-based algorithm to be used for arbitrary vector size achieving $O(l2^r(\log_2 n - 1)^r \log_2 n^{r+1})$ query time complexity [3].

2.2.2 Hierarchical Decomposition of the Search Space

For the $r \leq 1$ problem, Yao and Yao proposed a binary-search-like tree algorithm with $O(l \log\log n)$ query time complexity and $O(ln \log l)$ space complexity [42]. They first divide the query vector in two halves and search for the exact matches for each half. For each matched candidate, if it was found by only by one half of the query, their algorithm recursively apply this binary search strategy on the non-matching half until they reach the leaf of the tree (query vector has length 1).

For the general case, Brin [6] proposed a data structured called GNAT (Geometric Near-Neighbor Access Tree) that create a hierarchical decomposition of the search space and works with any metric distance included Hamming distance. Based on this data

[1]A machine word is the natural unit of data used by a particular processor. A word is a fixed-sized group of bits that are handled as a unit by the instruction set or processor hardware.

structure, Muja and Lowe [31] proposed an algorithm based on a k-medoids decomposition of the search space. K-medoids is an adaptation of the k-means algorithms that choose data points as centers, minimizing the sum of pairwise dissimilarities (distances) between the points in the cluster. A medoid can be defined as that member of a cluster whose average dissimilarity to all other members of the cluster is minimal. Intuitively it is the most centrally located member of the cluster.

The k-medoids algorithm uses a parameter k that is the number of clusters to form. Firstly it selects k random elements of the dataset as clusters center, and then it assigns the remaining dataset elements to the cluster where they are closest to its center. This process is repeated recursively inside every cluster until a maximum-leaf size is reached, forming a tree.

The algorithm of Muja and Lowe [31] repeats the k-medoid algorithm multiple times from different random cluster centers, creating multiple trees. The k-medoid tree is traversed starting from the root node. The child subtree with the closest center to the query is selected and the process is repeated on that that subtree. When a leaf is reached, all dataset elements on that node are scanned, using a linear search algorithm to find the Hamming distance to the query. This process is run in parallel on all trees. This algorithm does not give an exact answer and some r-neighbors to the query could be missed. The algorithm of Liu et al. [26] extends the idea of Muja and Lowe [31] by searching not only the query but perturbations of the query (variations of 1 bit, 2 bits, $\ldots$, p bits).

2.2.3 Locality-Sensitive Hashing by Random Projections

Locality-Sensitive Hashing (LSH) [20] is a widely used hashing technique based on a projection operator. A projection $proj : 2^l \to 2^n$ is a set of distinct numbers $k = \{p_0, p_1, \ldots p_{k-1}\}$ selected such as $0 \leq p_i < l$ and $n < l$. Using the projection, any vector $x \in 2^l$ is transformed to $proj(\mathbf{x}) \in 2^n$ where $proj(\mathbf{x})[i] = \mathbf{x}[p_i]$.

LSH methods are builds upon a simple idea: assume two binary vectors that are close in the Hamming space, then the probability of a projection of those vectors to be equal is very high. Using this idea, vectors are divided in subvectors which are stored in individual indices. Each index is queried proving a set of id (candidates) to be answer of the original query. That set of ids is linearly scanned to find the answer to the query.

Randomly Selected Subvectors

LSH methods based on a random selection of subvectors uses a set K of m random projections (called *keys* in this case). All subvectors $\mathbf{v}_i$ generated form a specific key $k_i \in K$ are stored in the hash table.

To search for the elements at distance r of the query $\mathbf{q}$, the same set of keys K is used to form subvectors of the query $q_1, q_2, \ldots, q_m$. Each subvector of the query is then searched in its corresponding hash table and all elements of the database having the same hash value are selected as candidates. The final result set for the query is computed from those selected as candidates by performing a linear scan and discarding those vectors with Hamming distance to the query greater than r. This last stage, known as the *verification stage*, has $O(n)$ time complexity with respect of the number of candidates.

Using the LSH with random selected subvectors, a single candidate can be retrieved multiple times form different hash tables. Hao et al. [17] take that into consideration to reduce the number of candidates to be verified. Their algorithm only considers as candidate those dataset vectors that have occurred at least a specific number of times (they call this parameter *collision count*). This algorithm significantly reduce the number of candidates in the verification stage, but it also discards lot of true candidates, decreasing the accuracy.

Moreover, in classic LSH with random selected subvectors, if none of the subvectors of the query is identical to those of a r-variation, this r-variation will not be retrieved as candidate in any hash table and then the algorithm will fail to include it as one of the solutions. This problem is known as *false rejection* (rejecting to select a true candidate) [13]. This probability of failure increases with the increase of the search distance r.

To address the above problem, Esmaeili et al. [13] proposed to also search for perturbations of the query subvectors. When searching for a query subvector in a hash table, they search also for variations of 1 bit, 2 bits, ..., p bits. This variation significantly reduces the number false rejections, but increases the amount of candidates to verify. The algorithm of Esmaeili et al. [13] avoids using a full scan for verification by assigning weight to every variation in decreasing order of the number of bit changes, and computing final candidate selection from those of high overall sum of weights.

Other way to address false rejection has been to select appropriately the keys instead of using random keys. In his Ph.D. thesis, Shakhnarovich [35] proposed to learn multiple semi-supervised hashing keys using boosting technique. Multiple hash keys are learned sequentially with boosting to maximize hashing accuracy of each hash key. The misshash samples in the current hash table will be penalized by large weights and then the algorithm uses the new weight values to learn the next hash key. Given a query, the true candidates missed from the active hash table are more likely to be found in the next hash table.

2.2.4 Locality-Sensitive Hashing by Non-Overlapping Subvectors

Instead of using random subvectors for applying the LSH method, it is possible to divide the vectors in m non-overlapping subvectors of $\lfloor \frac{l}{m} \rfloor$ or $\lceil \frac{l}{m} \rceil$ bits. Then, if two binary vectors differ in r bits, there are at least $p = m - \left\lfloor \frac{r}{\lfloor \frac{r}{m} \rfloor + 1} \right\rfloor$ subvectors that differ in at most $r' = \lfloor \frac{r}{m} \rfloor$ bits [43].

In the work of Liu et al. [26] they choose $m = r + 1$ and with that there will be at least one exact match (where Hamming distance is 0) since $r' = \left\lfloor \frac{r}{r+1} \right\rfloor = 0$ and $p = 1$. Manku et al. [28] choose $m = \left\lfloor \frac{r}{2} \right\rfloor + 1$ entailing at least $p = 1$ subvector with Hamming distance $r' <= 1$. Zhang et al. [43] proposed to use $m = \left\lfloor \frac{r+3}{2} \right\rfloor$ and defined tighter conditions for candidates to be valid derived from that selection of m.

One common problem of those approaches [28, 26, 43] is that they need to know in advance the radius parameter r in order to calculate the number of partitions m and create the hash tables. This could be a problem for applications requiring to query the dataset using different radius dynamically. Norouzi et al. [32] propose a method that can deal with dynamic search radius by fixing the number of partitions m and calculating the radius to search every subvector $r' = \left\lfloor \frac{r}{m} \right\rfloor$.

To deal with arbitrary r', Norouzi et al. [32] proposed to search not only in the bucket corresponding to the key in the hash table, but all buckets whose indices are within r' bits of the key bucket. That creates a compromise between the number of hash tables to use m (subvectors) and the performance of the algorithm.

To find all neighbors within a radius r of a binary vector of size l using only one hash table, all buckets whose indices are within r bits of the query bucket has to be examined. But, when splitting the vectors in m subvectors instead of searching $L(l, r)$ buckets in a single hash table, the number of buckets to search is:

$$m \times L\left(\frac{l}{m}, \left\lfloor\frac{r}{m}\right\rfloor\right). \tag{2.1}$$

It can be seen that there is a reduction in the number of buckets to search within when using $m > 1$ hash tables (see Equation 1.1).

Recall that the candidate vectors found from each index are merged into a single list of candidate answers to the query. This list of candidates is linearly scanned to find the answer to the original query. The size of the subvector determine the size of the candidate list to be verified, and m determines the number of subvectors constructed from a vector. m also specifies the number of indices created to store (and match) these subvectors. When m is small, each index contains more information; however, in this case, more hash buckets will be explored (Equation 1.1). For larger values of m, on the other hand, the number of buckets explored at each index is small (Equation 1.1). See [32] for details.

Finding a good value for m is central to the efficiency of multi-index hashing with non-overlapping subvectors. When the value of m is too large or too small the approach will not be effective, In the case of uniformly distributed codes, using the analytic cost model discussed in [32], the value of $m = \log_2(|D|)$ yields a near-optimal search cost. This method effectively reduces the number of bucket to search, but implies more memory consumption.

In their work, Zhang et al. [43] define an extended condition that can filter some false candidates before the verification stage. The set m equal to $\lfloor \frac{r+3}{2} \rfloor$, and their filtering mechanism uses the following two rules. When r is even, at least one subvector that is an exact match (with Hamming distance 0) or at least two subvectors with Hamming distance 1 are found in its index. When r is odd, valid candidates will have one subvector that is an exact match and one that is at most at Hamming distance 1, or at least three subvectors that are at Hamming distance 1. This condition reduces the number of candidates to be verified because having several conditions to fulfill effectively reduces the probability of false candidates progressing to the next stage. This is the only enhanced filtering technique that have been used to successfully reduce the number of candidate to be verified without removing true candidates.

The main problem with this condition is that the number of partitions m depends on knowing in advance the search radius, so this method is not viable for dynamic applications where searches with different radius are performed.

2.3 Summary

This chapter has presented a review of existing methods to solve the r-neighbors problem. As have been seen, existing trie techniques can only deal with very small radii (usually

$r \leq 1$). Although these methods are time-efficient, they are not practical for answering r-neighbors searches in very large databases due to their high storage requirement.

Methods based on hierarchical decomposition of the search space are not designed to address range queries in Hamming space. These methods are also not practical for very large databases due to their high query time complexity. Hashing based method are designed to deal with larger radius and larger database sizes. To reduce the complexity of dealing with large vectors, these methods divide the vectors in several smaller subvectors, which are used during the search. Search results obtained using subvectors are combined to construct the overall solution. Vectors can be divided using random projections or a non-overlapping subdivision. Random projections present the problem of false rejections when none of the subvectors of the query is identical to those of a valid candidate. Additionally, both schemes of dividing the vectors tend to generate lots of candidates to be tested during the verification stage, which is time consuming.

Hash table based methods that use non-overlapping subvectors need to construct r-variations for each subvector and match it in the hash table. While the number of r-variations for the set of subvectors computed from query vector is much smaller than the r-variations for the query vector itself, the total number of hash table lookups is too large, affecting the scalability of these methods. In the next chapter we will describe an approach that dramatically reduces the number of hash table lookups by using a hybrid, trie plus hash table, data structure.

Chapter 3

r-Neighbors Query Processing Using Hybrid Index Structures

In this chapter we describe two methods that use bitwise trie data structure to address the r-neighbors search problem (for arbitrary values of r). We first demonstrate that it is possible to identify r-neighbors of a query vector in a set of vectors stored in a bitwise trie. Traversing trie is time consuming, and trie based approach for resolving r-neighbors queries is slower when compared to existing techniques. This observation led us to propose a hybrid (trie + hash table) data structure that achieves state-of-the-art query processing times for r-neighbors queries.

3.1 Preliminaries

We begin by formally introducing r-variations.

3.1.1 r-Variations

Definition 9. *Given a binary vector* $\mathbf{q} \in 2^l$ *and a value* $r \in \mathbb{Z}^+$*, we define the* r-variations of $\mathbf{q}$ *denoted as* $Q(\mathbf{q}, r)$ *as all possible vectors of equal size of* $\mathbf{q}$ *at Hamming distance equal to* r *or less to* $\mathbf{q}$*:*

$$Q(\mathbf{q}, r) = \{\mathbf{q}' \in 2^l : H(\mathbf{q}, \mathbf{q}') \leq r\}$$

Example 3.1.1. *Given the binary vector* $\mathbf{q} = 0001$*, for the radius* $r = 2$ *the set of* r*-variations of* $\mathbf{q}$ *are:* $Q(\mathbf{q}) = \{1001, 0101, 0011, 0000, 1101, 1011, 1000, 0111, 0100, 0010\}$.

The definition of r-variations is closely related to the definition of the r-neighbors search problem. It is necessary to remember that this problem is defined as finding all vectors in D at Hamming distance r or less of a query $\mathbf{q}$ (See definition 8):

$$N_r(D, \mathbf{q}) = \{\mathbf{d} \in D : H(\mathbf{d}, \mathbf{q}) \leq r\}$$

The r-neighbors search problem can be also be defined as the r-variations that are in D:

$$N_r(D, \mathbf{q}) = Q(\mathbf{q}, r) \cap D$$

.

Those r-variations that are not in D are called *null variations*. Formally:

Definition 10. *Given a set of binary vectors* $D \subseteq 2^l$*, a binary vector* $q \in 2^l$ *and a value* $r \in \mathbb{Z}^+$*, all vectors* $x \in Q(q, r) \setminus N_r(D, q)$ *are called* null variations.

Example 3.1.2. *Given the set of binary vectors* $D = \{1011, 1010, 1000\}$*, the answer of the* r*-neighbors search problem for* $\mathbf{q} = 0001$ *and* $r = 2$ *is* $N_2(D, \mathbf{q}) = \{1011, 1000\}$ *(see Example 2.1.7). Those vectors are* 2*-variations of* $\mathbf{q}$ *that are in* D *(see example Example 3.1.1), the other* 2*-variations of* $\mathbf{q}$*, vectors* $1001, 0101, 0011, 0000, 1101, 0111$ *and* 0010 *are all* null variations.

The reader should notice that the number of variations is directly dependent on both, the length of the vector and the radius. The total number of r-variations of a vector $\mathbf{q}$ is equal to:

$$|Q(\mathbf{q}, r)| = \sum_{i=0}^{r} \binom{|\mathbf{q}|}{i}$$

3.1.2 Computing All r-Variations

To compute the set of r-variations of a vector $\mathbf{q}$, the followed recursive approach can be followed: we generate all prefixes of $\mathbf{q}$ that is at Hamming distance 1 and then the process is recursively repeated for the remaining suffix of $\mathbf{q}$ but this time with using radius $r - 1$. Formally it can be defined as a recursive expression:

Base cases:

$$Q(\mathbf{q}, 0) = \{\mathbf{q}\}$$

$$Q(\epsilon, r) = \{\epsilon\}$$

Inductive case:

$$Q(\mathbf{q}, r) = \{\mathbf{q}[0] \| \mathbf{v} : \mathbf{v} \in Q(\mathbf{q}[1:], r)\} \cup \{\neg\mathbf{q}[0] \| \mathbf{v} : \mathbf{v} \in Q(\mathbf{q}[1:], r-1)\}$$

Algorithm 3.1 shows the pseudo-code for generating the r-variations using the above recursive expression.

Algorithm 3.1 Algorithm to generate all possible r-variations of a given vector

```
procedure GENERATEALLVARIATIONS($\mathbf{q}, r$)
    $V \leftarrow \emptyset$
    if $r = 0$ then
        return $\{\mathbf{q}\}$
    if $|\mathbf{q}| = 0$ then return $\{\epsilon\}$
    else
        $p \leftarrow \mathbf{q}[0]$
        $\mathbf{s} \leftarrow \mathbf{q}[1:]$
        for all $\mathbf{s}' \in$ GENERATEVARIATIONS($\mathbf{s}, r$) do
            $V \leftarrow V \cup \{p \| s'\}$
        for all $\mathbf{s}' \in$ GENERATEVARIATIONS($\mathbf{s}, r-1$) do
            $V \leftarrow V \cup \{(\neg p) \| \mathbf{s}'\}$
    return $V$
```

3.2 Solving the r-Neighbors Search Problem Using a Trie

In this section we are going to present a trie based index structure to solve the r-neighbors search problem. This index structure is inspired by the way Algorithm 3.1 works. The goal of this index structure is to find all non-null r-variations of a given query. Algorithm 3.1 works by fixing a prefix of Hamming distance k to $\mathbf{q}$ and then generating all possible $(r-k)$-variations that have that prefix. At every recursive call it adds one more value to that fixed prefix.

A trie [11] is a tree data structure where all the descendants of a node have a common prefix of the vector (string) associated with that node, and the root is associated with the empty vector (string). Each node at level l represents the set of all keys that begins with the same sequence of l characters called its prefix; the node defines a branching depending on the $l + 1$ character of the keys. The trie for the special case of the binary vectors (called *bitwise trie*) is a binary tree since the values at each position are 0 or 1. Figure 3.1 shows an example of a bitwise trie.

A bitwise trie can be formally defined as:

Definition 11. *Given $D \subset 2^l$. A* bitwise trie *of D denoted as T_D is defined as follow:*

i) All nodes $n \in T_D$ represent a subset of vectors of D denoted as $V(n)$ that share a common prefix $p(V(n))$.

ii) All node $n \in T_D$ is divided in two subtree named $left(n)$ and $right(n)$ (see Figure 3.2):

- *The left subtree represents a subset of the vectors associated with n that has a bit equal zero in the position after the common prefix. Formally $V(left(n)) = \{\mathbf{x} \in V(n) : \mathbf{x}[|p(V(n))|] = 0\}.n$*
- *The right subtree represents a subset of the vectors associated with n that has a bit equal one in the position after the common prefix. Formally $V(right(n)) = \{\mathbf{x} \in V(n) : \mathbf{x}[|p(V(n))|] = 1\}.$*

iii) For all nodes n, the length of its prefix is one bit less than the length of the prefix of

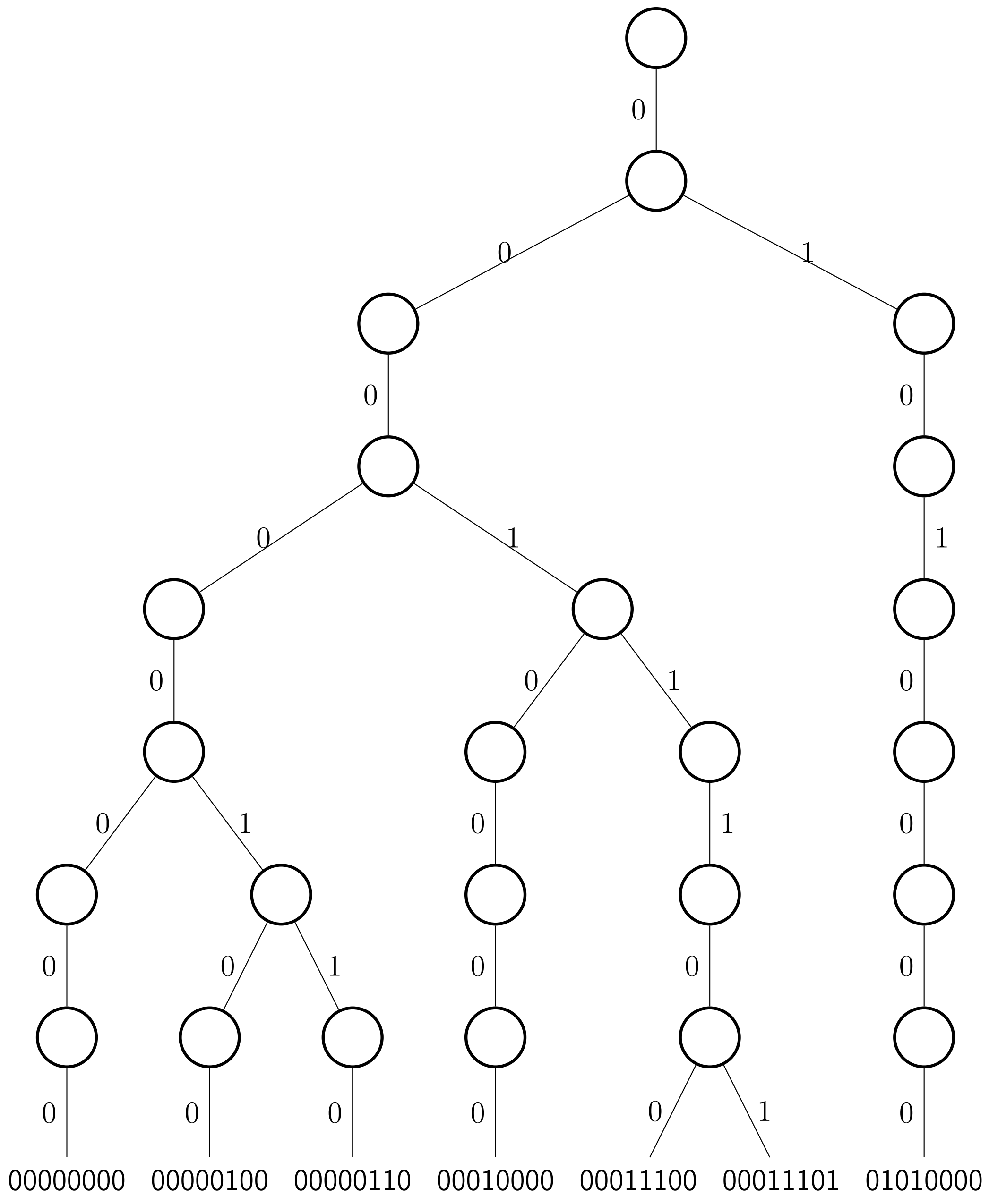

Figure 3.1: Bitwise trie for $D = \{$ 01010000, 00010000, 00011100, 00011101, 00000110, 00000100, 00000000 $\}$

each subtree. Formally $|p(V(n))| = |p(V(left(n)))| + 1 = |p(V(right(n)))| + 1.$

Nodes with the two children equal to the empty set are called leaf *nodes. Nodes with at least one non-empty child are called* tree *nodes.*

The *root* node corresponds to the whole set D. Each tree node corresponds to the

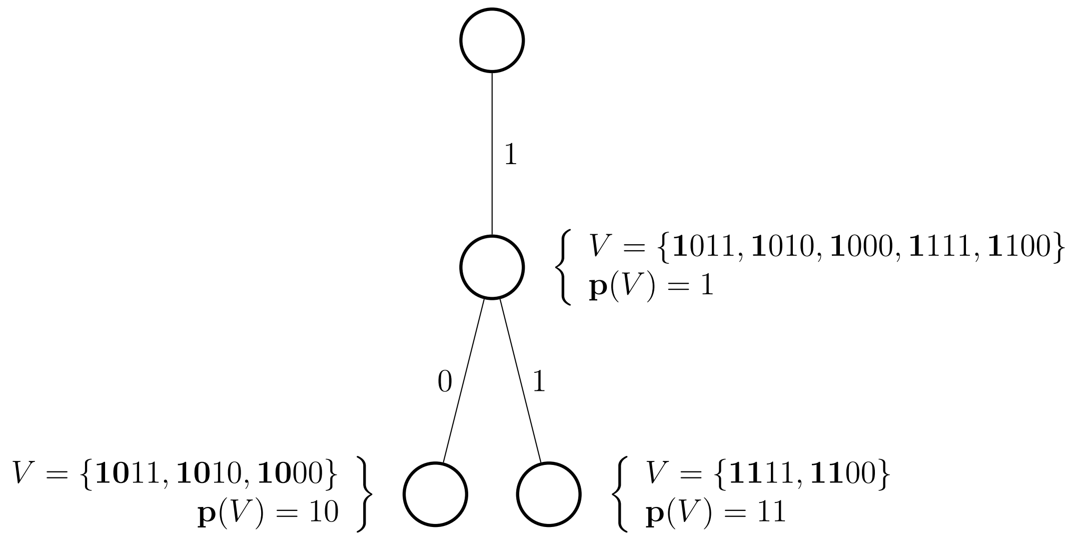
1
$V = \{\mathbf{1}011, \mathbf{1}010, \mathbf{1}000, \mathbf{1}111, \mathbf{1}100\}$
$\mathbf{p}(V) = 1$
0
1
$V = \{\mathbf{10}11, \mathbf{10}10, \mathbf{10}00\}$
$\mathbf{p}(V) = 10$
$V = \{\mathbf{11}11, \mathbf{11}00\}$
$\mathbf{p}(V) = 11$

Figure 3.2: Node child splitting example on a bitwise trie.

common prefix of its leaf nodes. Furthermore, the common prefix of a child is strictly longer than that of its parent. Thus, the length of the trie built from $D \subset 2^l$ must be at most (exactly) l.

Given a bitwise trie T_D, it is possible to know if a vector x is in D by traversing the trie. If a leaf is reached, $x \in D$, otherwise $x \notin D$. Algorithm 3.2 shows the pseudo-code to query a bitwise trie.

Using a similar idea to Algorithm 3.2, we can traverse the trie searching for r-variations. In this case, at every node we are going to visit both children keeping track of the hamming distance of the maximum common prefix of the visited node with the query. However, since a child of an internal node can be empty, no non-null r-variation can be generated from that child, which is an effective pruning of null r-variations. Since an r-variation is reported when reached a leaf, no null $r-variation$ is generated. Algorithm 3.3 shows the pseudo-code for generating all non-null r-variations of $\mathbf{q}$ given T_D:

Lemma 1. *Algorithm 3.3 does not generate null variations.*

Algorithm 3.2 Algorithm to query a Bitwise Trie

```
procedure SEARCH($T_D$, $\mathbf{q}$)
    if $T_D = \emptyset$ then
        return False
    $node \leftarrow root(T_D)$
    $level \leftarrow 0$
    while not $isLeaf(node)$ do
        if $\mathbf{q}[level]) = 1$ then
            $node \leftarrow rightChild(node)$
        else
            $node \leftarrow leftChild(node)$
        $level \leftarrow level + 1$
    if $isLeaf(node)$ then
        return True // Found
    else
        return False // Not found
```

Proof. Since Algorithm 3.3 only return vectors when reaching a leaf node (line 10), all variations returned are in D. □

3.2.1 Reducing Space Requirement Using Compressed Bitwise Trie

An space-optimized variant of the trie (called compressed trie) is where each node with only one child is merged with its parent. This way, all internal nodes have at least 2 children.

For this particular case, E. G. Coffman and J. Eve [9] propose to store at each internal node the first bit position where its vectors differs. All vectors containing a 1 at that position will be in the right child and all with 0 in the left child. This type of trie is what we will call *compressed bitwise trie*. Formally speaking:

Definition 12. *Given $D \subset 2^l$. A* compressed bitwise trie *of D denoted as CT_D is defined*

Algorithm 3.3 r-neighbor search using a trie

```
procedure RANGEQUERY($T_D, \mathbf{q}, r$)
    if $T_D = \emptyset$ then
        return $\emptyset$
    return RANGEQUERYAT($root(T_D), 0, \mathbf{q}, r$)

procedure RANGEQUERYAT($node, i, \mathbf{q}, r$)
    if $node = NIL$ or $r < 0$ then
        return $\emptyset$
    if $isLeaf(node)$ then
        return $\{\mathbf{q}\}$
    else
        if $\mathbf{q}[i] = 0$ then
            $VariationsLeft \leftarrow$ RANGEQUERYAT($left(node), i+1, \mathbf{q}, r$)
            $VariationsRight \leftarrow$ RANGEQUERYAT($right(node), i+1, \mathbf{q}, r-1$)
        else
            $VariationsRight \leftarrow$ RANGEQUERYAT($right(node), i+1, \mathbf{q}, r$)
            $VariationsLeft \leftarrow$ RANGEQUERYAT($left(node), i+1, \mathbf{q}, r-1$)
        return $VariationsLeft \cup VariationsRight$
```

as follow:

i *All nodes $n \in CT_D$ represent a subset of vectors of D denoted as $V(n)$ that share a common prefix $p(V(n))$. The value $|p(V(n))|$ is called* split position *(splitPos).*

ii *When the split position of a node is l, it is a leaf node, otherwise it is an internal node.*

iii *All internal nodes $N \in CT_D$ have two subtree named $left(N)$ and $right(N)$.*

- *The left subtree represents a subset of the vectors associated with n that has a bit equal zero in the position after the common prefix. Formally $V(left(n)) = \{\mathbf{x} \in V(n) : \mathbf{x}[|p(V(n))|] = 0\}$.*
- *The right subtree represents a subset of the vectors associated with n that has a bit equal one in the position after the common prefix. Formally $V(right(n)) =$*

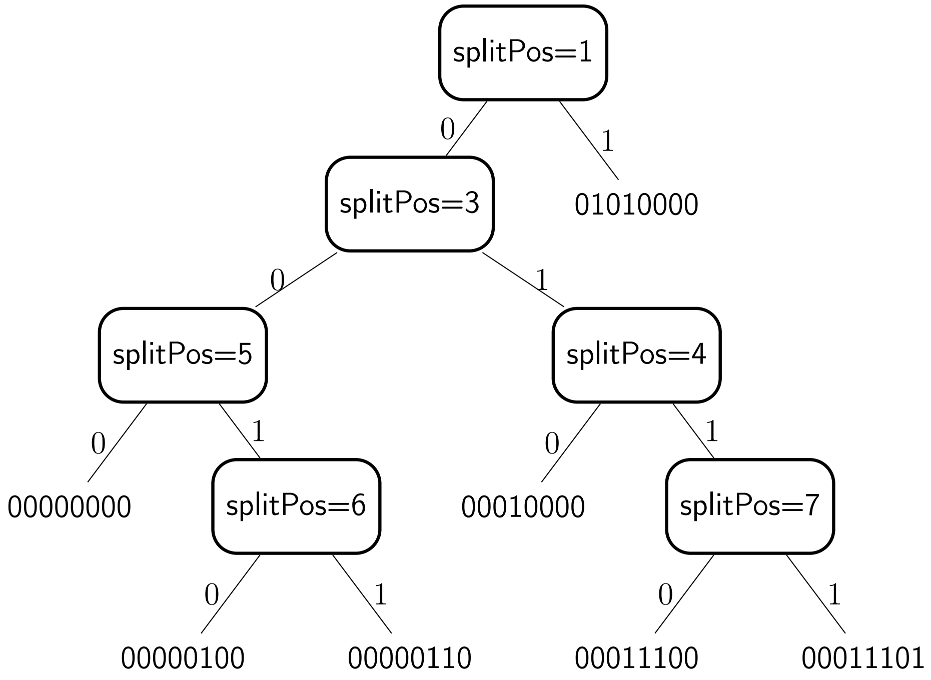

Figure 3.3: Compressed bitwise trie for D = { 01010000, 00010000, 00011100, 00011101, 00000110, 00000100, 00000000 }

$$\{\mathbf{x} \in V(n) : \mathbf{x}[|p(V(n))|] = 1\}.$$

Figure 3.3 shows an example of the compressed bitwise trie.

Algorithm 3.3 can be revised to proces r-neighbors queries using a compressed bitwise trie (see Algorithm 3.4).

In a compressed trie, a pair of parent-child nodes could represent a compression of several nodes in the trie that has only one child. It is obvious that the only bits with two options to explore when searching for r-variations are those at the split position, but the query vector could have different bit values of those bit that has only one child. For example, if the common prefix of a node is 01, and the query is 111, when the split position of that node is 2, the first value of the query (0) differs from the prefix. Algorithm 3.4 uses a *lazy evaluation strategy* since it does not verify that all bits of the

Algorithm 3.4 Lazy-evaluation algorithm for r-neighbor search using a compressed bitwise trie

```
1: procedure RANGEQUERY($CT_D, \mathbf{q}, r$)
2:     if $T_D = \emptyset$ then
3:         return $\emptyset$
4:     return RANGEQUERYAT($root(CT_D), \mathbf{q}, r$)
5: procedure RANGEQUERYAT($node, i, \mathbf{q}, r$)
6:     if $node = NIL$ or $r < 0$ then
7:         return $\emptyset$
8:     if $isLeaf(node)$ then
9:         if $H(\mathbf{q}, V(node)) \leq r$ then
10:            return $\{\mathbf{q}\}$
11:    else
12:        if $\mathbf{q}[splitPos(node)] = 0$ then
13:            $VariationsLeft \leftarrow$ RANGEQUERYAT($left(node), \mathbf{q}, r$)
14:            $VariationsRight \leftarrow$ RANGEQUERYAT($right(node), \mathbf{q}, r-1$)
15:        else
16:            $VariationsRight \leftarrow$ RANGEQUERYAT($right(node), \mathbf{q}, r$)
17:            $VariationsLeft \leftarrow$ RANGEQUERYAT($left(node), \mathbf{q}, r-1$)
18:        return $VariationsLeft \cup VariationsRight$
```

query match the common prefix (bit 0 in our example). Those bits will be checked once a leaf is reached.

Algorithm 3.5 is a non-lazy evaluation version of the Algorithm 3.4. The main difference between those two algorithms is that the second algorithm verifies the full prefix of the query before flipping the bit at the split position.

3.3 Hybrid Trie-Hash Table Index Structure for the r-Neighbors Search Problem

There are two main group of algorithms for solving the r-neighbors search problem. On one hand, Hash-based r-neighbors search methods uses a algorithm to generate all r-

Algorithm 3.5 Algorithm for r-neighbor search using a compressed bitwise trie

```
1: procedure rangeQuery($CT_D, \mathbf{q}, r$)
2:     if $T_D = \emptyset$ then
3:         return $\emptyset$
4:     return rangeQueryAt($root(CT_D), \mathbf{q}, r$)
5: procedure rangeQueryAt($node, i, \mathbf{q}, r$)
6:     if $node = NIL$ or $r < 0$ then
7:         return $\emptyset$
8:     if $isLeaf(node)$ then
9:         return $\{\mathbf{q}\}$
10:    $r \leftarrow r - H(\mathbf{q}[: splitPos(node)], p(V(node)))$
11:    $\mathbf{q} \leftarrow p(V(node)) \| \mathbf{q}[splitPos(node) :]$
12:
13:    if $q[splitPos(node)] = 0$ then
14:        $VariationsLeft \leftarrow$ rangeQueryAt($left(node), \mathbf{q}, r$)
15:        $\mathbf{q}[splitPos(node)] \leftarrow 1$
16:        $VariationsRight \leftarrow$ rangeQueryAt($right(node), \mathbf{q}, r - 1$)
17:    else
18:        $VariationsRight \leftarrow$ rangeQueryAt($right(node), \mathbf{q}, r$)
19:        $\mathbf{q}[splitPos(node)] \leftarrow 0$
20:        $VariationsLeft \leftarrow$ rangeQueryAt($left(node), \mathbf{q}, r - 1$)
21:    return $VariationsLeft \cup VariationsRight$
```

neighbors and use a hash table to check if they are on D. Checking a hash table is relatively inexpensive, but the exponential number of r-variations to check, most of them null r-variations, makes those algorithms computationally expensive for large vector sizes and large radii.

On the other hand, trie based algorithm traverses the trie avoiding generating null r-variations. The major disadvantage of this method is that when r reaches 0 during trie traversal, the traversal must continue to see if non-null r-variation exists in the trie. This adversely affects the performance of the overall algorithm. Figure 3.4 highlights this observation. Paths shown in blue refer to traversals after r reaches 0. Clearly, there are many more blue paths than there are red paths, which refer to traversal corresponding to $r > 0$.

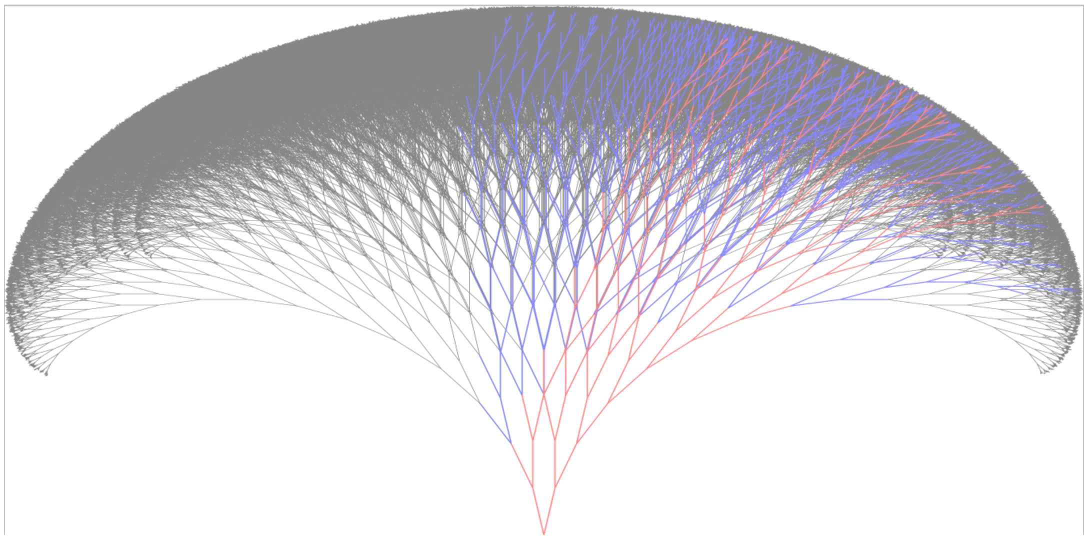

Figure 3.4: This image shows the nodes traversed when $r > 0$ (in red) and when $r = 0$ in blue on a trie of 1 million 64 bit vectors searched with $r = 2$. This suggests that a mechanism to avoid traversal when $r = 0$ is reached may lead to significant performance gain.

We have developed a novel strategy that uses a hash table to avoid subsequent traversals when $r = 0$ is reached. This results in a substantial reduction in the number of nodes explored during candidate r-variations generation. The proposed scheme stores the collection of binary vectors in both a trie and a hash table. Given a query vector, trie data structure is used to generate the candidate r-variations. This is accomplished by traversing the trie. During traversal, when r reaches 0, the current candidate is looked up in the hash table; we refer to it as the *membership check*. This determines in constant time if the current r-variation candidate is in the collection. No further traversals are needed below this node. Choosing hash table lookup at $r = 0$ is simply a design decision. It is indeed possible to extend the idea of using hash table for membership checks to $r > 0$; however, we would like to remind the reader the exponential growth of membership checks for values of r greater than 0 (see Equation 3.1.1). Thus choosing r $>$ 0 may result in little or no performance gain.

The hybrid index structure consist of a hash table HT and a compressed bitwise trie CT. All elements of D will be inserted in both structures. Algorithm 3.6 shows the pseudo-code to query the hybrid index. The reader should notice that when comparing the function rangeQueryAt between Algorithm 3.5 and Algorithm 3.6, the only significant change is that in the later recursion terminates when $r = 0$ (lines 13 and 14 of Algorithm 3.5).

Algorithm 3.6 Algorithm for r-neighbor search using hybrid index

```
1:  procedure rangeQuery($HT_D, CT_D, \mathbf{q}, r$)
2:      if $T_D = \emptyset$ then
3:          return $\emptyset$
4:      $Variations \leftarrow$ rangeQueryAt($root(CT_D), \mathbf{q}, r$)
5:      $Result \leftarrow \emptyset$
6:      for all $\mathbf{v} \in Variations$ do
7:          if $\mathbf{v} \in HT_D$ then
8:              $Result \leftarrow Result \cup \{\mathbf{v}\}$
9:      return $Result$
10: procedure rangeQueryAt($node, i, \mathbf{q}, r$)
11:     if $node = NIL$ or $r < 0$ then
12:         return $\emptyset$
13:     if $isLeaf(node)$ or $r = 0$ then
14:         return $\{\mathbf{q}\}$
15:     $VariationsAtNode \leftarrow \emptyset$
16:     if $H(\mathbf{q}[: splitPos(node)], p(V(node))) > 0$ then
17:         $r \leftarrow r - H(\mathbf{q}[: splitPos(node)], p(V(node)))$
18:         $q \leftarrow p(V(node)) \| \mathbf{q}[splitPos(node) :])$
19:     if $q[splitPos(node)] = 0$ then
20:         $VariationsLeft \leftarrow$ rangeQueryAt($left(node), \mathbf{q}, r$)
21:         $\mathbf{q}[splitPos(node)] \leftarrow 1$
22:         $VariationsRight \leftarrow$ rangeQueryAt($right(node), \mathbf{q}, r - 1$)
23:     else
24:         $VariationsRight \leftarrow$ rangeQueryAt($right(node), \mathbf{q}, r$)
25:         $\mathbf{q}[splitPos(node)] \leftarrow 0$
26:         $VariationsLeft \leftarrow$ rangeQueryAt($left(node), \mathbf{q}, r - 1$)
27:     return $VariationsLeft \cup VariationsRight$
```

3.3.1 Multi-Hybrid Index

The hybrid index structure is proposed as a replacement of a pure hash table index. However, for large vectors and large radius r, the best approach is to use a multi-index schema like the *locality-sensitive hashing* methods (see Section 2.2.3). Since the the work of Norouzi et al. [32] is the best algorithm to date that deal with the dynamic r-neighbors problem, we propose to use their scheme but using our hybrid index. Instead of using only a hash table, each index in the multi-index scheme would be a hybrid trie + hash table data structure.

The multi-index approach using the proposed hybrid index structure is as follows. Given $D \in 2^l$ and $m < l$:

1. Vectors of D are divided in non-overlapping subvectors $\mathbf{v}_1, \mathbf{v}_2, \dots \mathbf{v}_m$ of size $\lfloor \frac{l}{m} \rfloor$ or $\lceil \frac{l}{m} \rceil$ such as $\sum_{i=1}^{m} |\mathbf{v}_i| = l$ (see Section 2.2.4).

2. m hybrid indexes are created denotes as $I_1, I_2, \dots I_m$, and on each one is inserted one chunk of each subvector of D (all $\mathbf{v}_1$ are inserted in I_1, all $\mathbf{v}_2$ are inserted on I_2, etc)

To search for a query $\mathbf{q}$ and radius r using the multi-index, $\mathbf{q}$ it is also divided in $\mathbf{q}_1, \mathbf{q}_2, \dots \mathbf{q}_m$ in the same way vectors of D were divided. Then each subvector of the query is searched on the corresponding index ($\mathbf{q}_1$ is searched on I_1, $\mathbf{q}_2$ is searched on I_2, etc), but using radius $r' = \lfloor \frac{r}{m} \rfloor$. The result on every index is then tested to see if its corresponding vector on D is at Hamming distance r of the query $\mathbf{q}$ (see Section 2.2.4).

Chapter 4

Experimental Evaluation

Norouzi et al. [32] multi-index hashing implementation is available to the research community. Our compressed trie index implementation is added to their system. This enables us to compare our algorithm with theirs in a fair manner. From hence forth we refer to their algorithm as MIH; whereas, our technique that uses both a hash table and a compressed trie to index binary vectors is referred as MIH+Trie.

Experiments are run on a workstation with a 2.9 GHz quad-core Intel Xeon processor, 20 MB of L2 cache, and 64 GB of RAM. It is worth noting that large L2 cache significantly improves the performance of linear scan [32]. For our experiments, we only used a single core to simplify runtimes measurements. The runtimes reported in this work are the result of five runs of the algorithm in exactly the same conditions. The datasets used for evaluation are uniformly-distributed randomly-generated vectors.

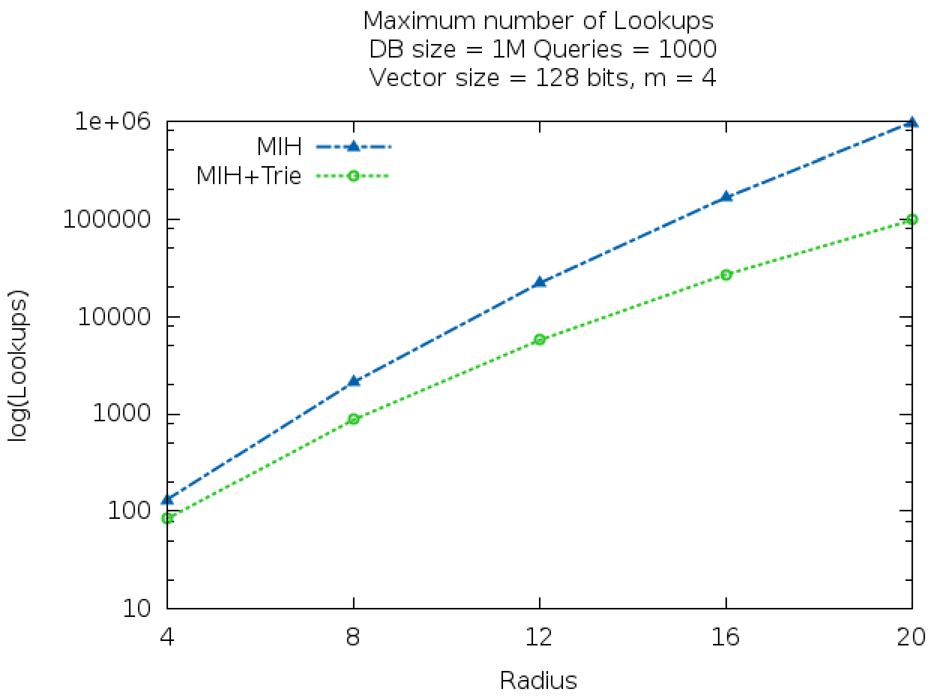

Figure 4.1: Number of hash table lookups for MIH and our hybrid approach (MIH+Trie) for a database of 1 million 128-bits vectors and using m=4 indexes. The number of lookups is an average over 1000 queries. Here every index manages vectors of 32 bits.

4.1 Hash Table Lookups

Since the goal of our hybrid approach is to improve runtime by reducing the number of hash table lookups that need to be performed, we compare both methods on the number of hash table lookups performed. MIH uses the naïve approach to compute all r-variations, and the total number of lookups that it will perform can be computed using Equation 1.1 for every block (parameter m). MIH+Trie, however, skips null r-variations, reducing the number of lookups to perform.

Figure 4.1 shows the difference in terms of the number of lookups performed between these two methods for a dataset of one million 128-bits vectors using 4 indexes (parameter m). In this scenario, each index will deal with $128/4 = 32$ bits vectors. Notice that the

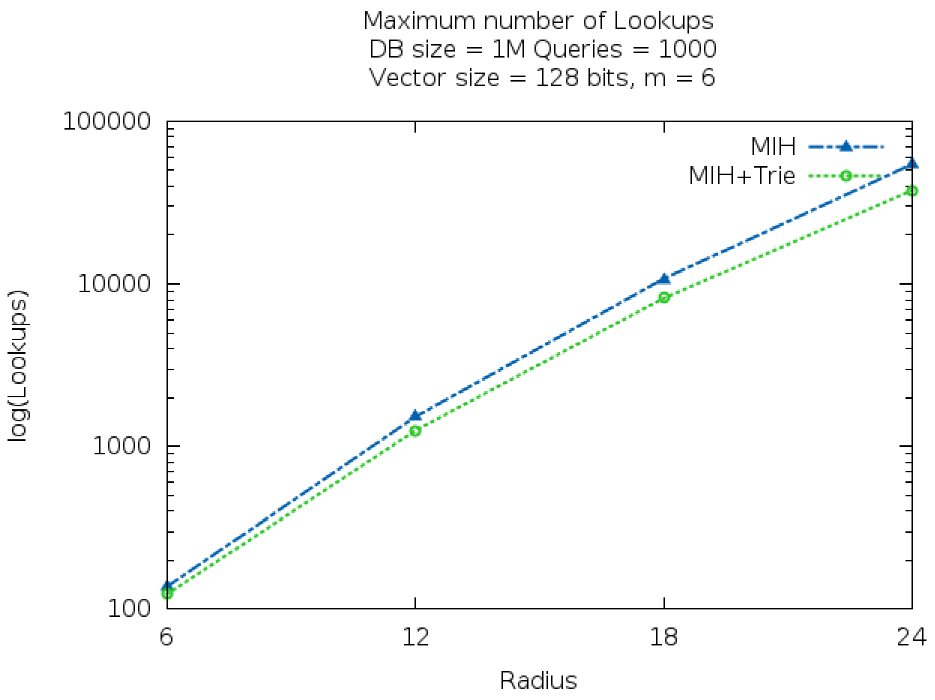

Figure 4.2: Number of hash table lookups between MIH and our hybrid approach (MIH+Trie) for a database of 1 million 128-bits vectors and using m=6 indexes. The number of lookups is an average over 1000 queries. Here every index manage vectors of 21 bits.

number of hash table lookups for the propose approach, MIH+trie, is much smaller than the lookups for MIH. Furthermore, this differences increases quickly for increasing values of r.

Figure 4.2 repeats the experiment with 6 indexes. $m = 6$ is the theoretical "best" value for this scenario as determined by MIH [32]. Again, observe the hash table lookups savings obtained by our method over MIH. This time, however, the savings obtained are not as good as those obtained in the previous case ($m = 4$). When using 6 indexes to store 128 bit vectors, each index stores $128/6 \approx 21$ bit vectors. The total number of 21 bit vectors is 2^{21}, which is $\approx 10^6$. Since we are indexing 1 million vectors, the trie corresponding to each index are nearly full. When trie is full, it looses its ability to prune

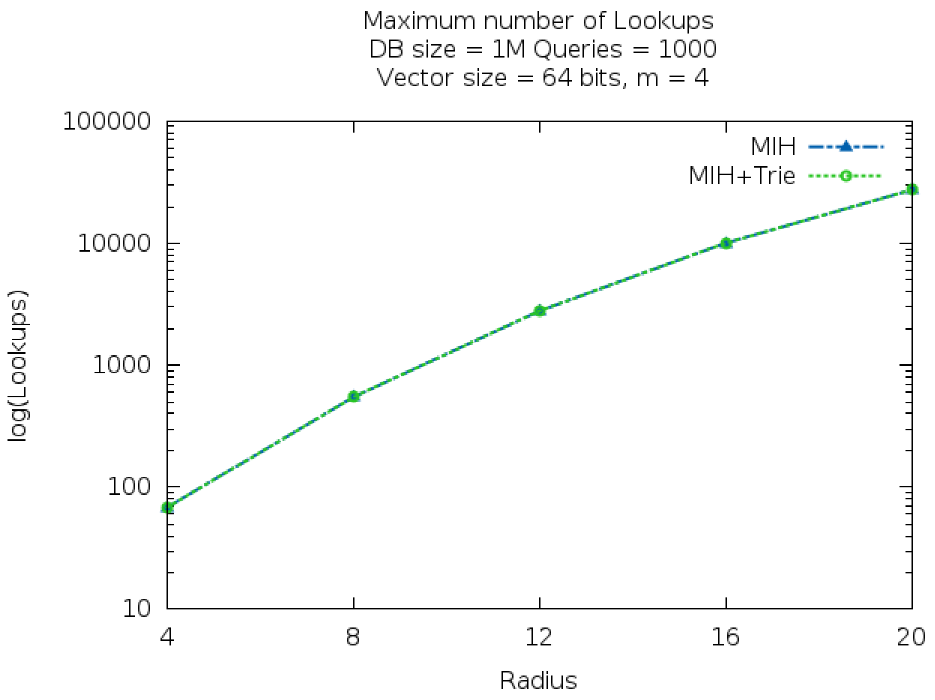

Figure 4.3: Comparison in the number of lookups between MIH and our hybrid approach (MIH+Trie) for a database of 1 million 64-bits vectors and using m=4 indexes. The number of lookups is an average over 1000 queries. Here every index manages vectors of 16 bits.

null r-variations—indeed, there are no null r-variations in this case. Consequently, every r-variation will be checked in the hash table. In such cases, our hybrid index is actually slower due to the extra processing costs associated with generating r-variations using a trie.

Figure 4.3 illustrates this issue further. Here we index 1 million 64 bit vectors using 4 indexes. Each index, therefore, stores (64/4 =)16 bit vectors. The total number of 64 bit vectors is 65536($\ll 10^6$total number of vectors). Each trie stores 1 million 16-bit vectors. In this case each trie is almost full. Consequently there is no null r-variations to prune, so both method performs the same number of lookups. This suggests that the choice of m is an important consideration when our method. A large value of m can

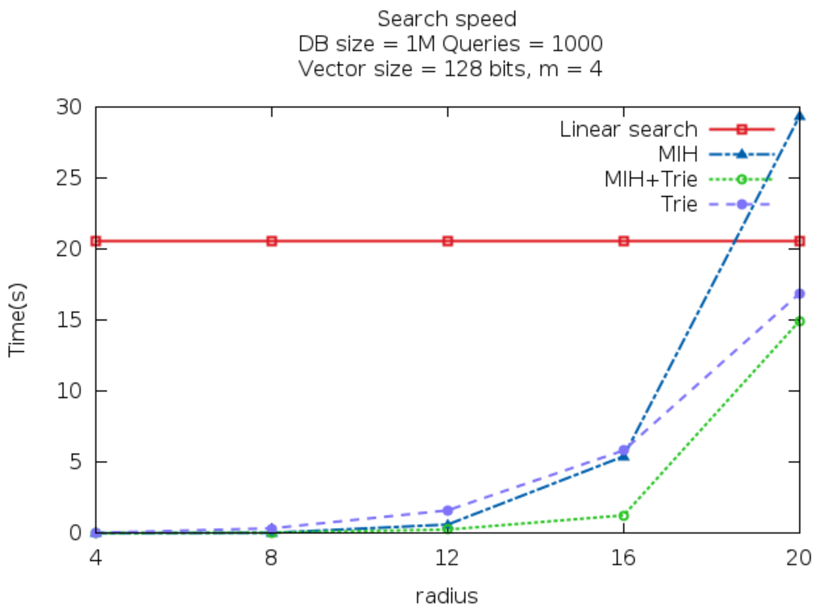

Figure 4.4: Comparison in time between Linear scan, Trie, MIH and our hybrid approach (MIH+Trie) for a database of 1 million 128-bits vectors, 1000 queries, using m=4 indexes. Here every index manages vectors of 32 bits.

make the tries dense while small values of m can makes them sparse tries.

4.2 Runtime Comparison

In the previous section (Section 4.1) we studied the number of hash table lookups generated by our method in comparison with the number of lookups performed by MIH. It was observed that, when the tries are not full, there is a reduction in the number of hash table lookups generated by our method. In this section we want to explore if there is empirical evidence that this reduction implies improved runtimes.

Figure 4.4 shows a comparison of the runtime on 1000 queries for the linear scan method (that serves as baseline), the MIH scheme, and our method MIH+Trie. The linear scan method does not depend on the radius, while both MIH and MIH+Trie

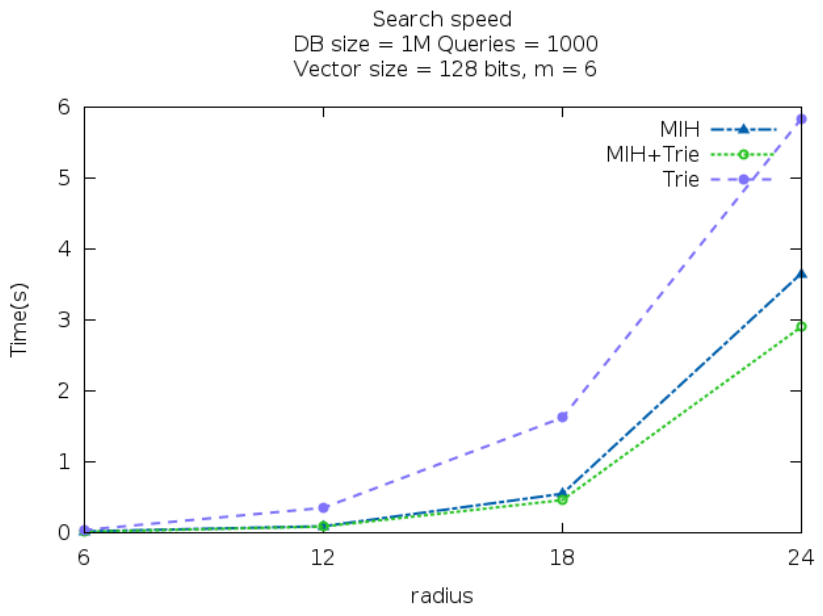

Figure 4.5: Comparison in time between Trie, MIH and our hybrid approach (MIH+Trie) for a database of 1 million 128-bits vectors, 1000 queries and using m=6 indexes. Here every index manages vectors of 21 bits.

methods depend upon r. Specifically, the processing times for all three increase sharply for increasing r. This behavior is to be expected. As can be seen, MIH+Trie outperforms both methods, while the MIH method perform worse than the linear scan for radii greater than 20 in this scenario.

The difference in speed is closely related to the difference in the number of hash table lookups performed. When the 64-bits vectors are divided into 6 indexes (i.e., $m = 6$), the difference in the number of hash table lookups is less than (Figure 4.2) the lookups when the vectors are divided into 4 indexes (Figure 4.1). This reduction in the difference of the number of hash table lookups dramatically reduces the difference in runtime between the two method. Even when MIH+Trie outperforms MIH for all radii, the difference in speed is only noticeable on very large radius ($r \geq 24$) as shown in Figure 4.5.

When both methods perform the same number of hash table lookups (see Figure 4.3),

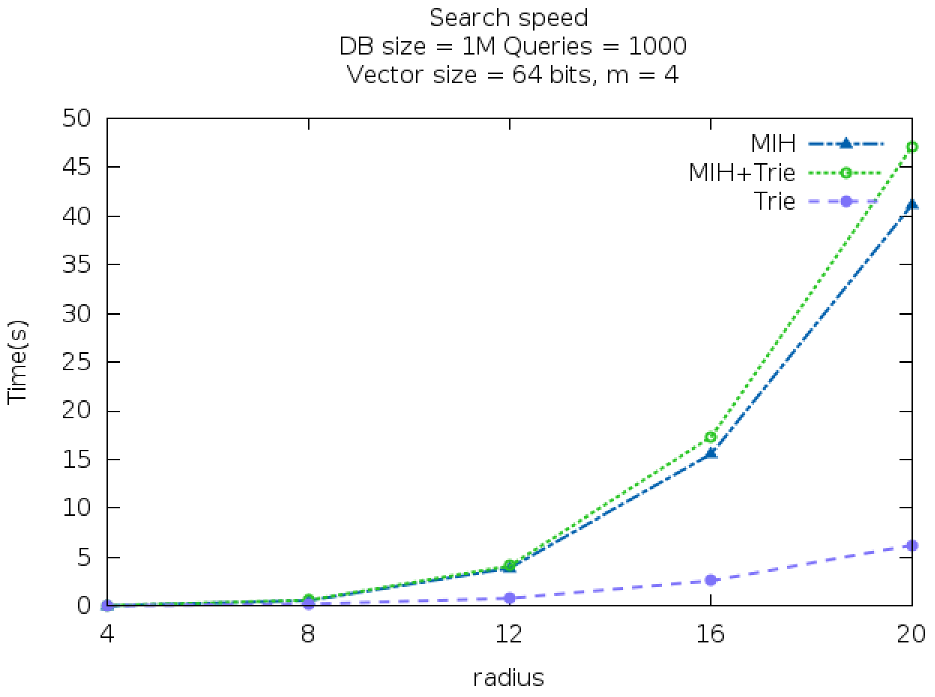

Figure 4.6: Comparison in time between Trie, MIH and our hybrid approach (MIH+Trie) for a database of 1 million 64-bits vectors, 1000 queries and using m=4 indexes. Here every index manages vectors of 16 bits.

our method performs worse than the MIH method, see Figure 4.6. This is due to the fact that trie is slower at generating r-variations.

4.3 Practical Considerations

In Section 4.1, we saw the reduction in the number of lookups performed by our method in comparison with a pure hash table when the trie is not full. On sparse tries, this reduction can be of several orders of magnitude as Figure 4.1 shows. In Section 4.2 we get that this reduction of lookups implies a substantially reduction in the search time (see Figure 4.4).

However, in a full trie there is no reduction in the number of lookups (see Figure 4.3).

When generating the lookups using the trie traversal, there is an additional overhead. This overhead, without accompamying reduction in the number of lookups, makes the runtime of our algorithm is worse than a pure hash method when the trie is full (see Figure 4.6).

This observations are empirical evidences that shows that there is a positive correlation between the number of lookups generated and the runtime of the algorithm. Traversing the trie to generate lookups has an overhead, but this is much smaller than the performance gain from avoiding hash lookups. This re-enforces our design principle of minimizing the number of lookups to increase search speed.

Practically, the trie will be very sparse for large vectors even with very large datasets. For example, a trie of a dataset of 1-billions 64-bits vectors is much less than 1% full. Therefore, as our experiments indicates, our approach outperforms the MIH for real-life datasets.

Chapter 5

Conclusion

In this thesis, we have studied the r-neighbor search problem within a finite radius in high dimensional Hamming space. Formally, the problem is defined as: given a collection, D, of binary vectors, a query $\mathbf{q}$ which is also a binary vector, and a radius $r > 0$, find all vectors in X that is at most r distance away from $\mathbf{q}$. Namely, $\{\mathbf{x} \in X : H(\mathbf{x}, \mathbf{q}) \leq r\}$.

The state-of-art solution to our problem in the existing literature is the multi-index hashing (MIH) approach proposed by Nozouzi et al. [32]. We have identified that the bottleneck of using MIH index structure is the exponentially many hash lookups that must be generated when searching for r-neighbors. The effect of the bottleneck is exasperated with longer vector length (namely the dimensionality of the Hamming space). The root cause of the bottleneck is that the hashing nature of MIH does not support local search.

5.1 Algorithms

Our proposal is to augment the hash-based MIH data structure so that it is possible for a search algorithm to perform limited local search, reducing the number of hash lookups needed. To this end, we augment the MIH index with a compressed trie. A compressed trie is a data structure that organizes the binary vectors according to their common prefixes. The compressed trie offers two important features that alleviate the bottleneck of MIH:

1. the organization by common prefix allows efficient local search (by prefix), thus it can guide the search by pruning. This allows much faster convergence to the final query answer.

2. as part of the compression for storage, compressed trie records important bits that partitions the dataset into two nonempty sets. Unlike the naïve candidate query generation used in MIH, we are able to limit the bit-flipping to only these important bits. This significantly reduces the number of hash lookups needed.

We have formalized the improved efficiency by introducing the concepts of r-variations of a query $\mathbf{q}$ and null r-variations of $\mathbf{q}$ with respect to a dataset D. An r-variation of $\mathbf{q}$ is a candidate query that is generated by MIH, while a null r-variation is one that fails to locate any neighbors in D. So, all generated null r-variations are undesirable as they are wasted computations. A key difference between the naïve MIH and our approach is that we are able to control the amount of null r-variations generated by the algorithm.

In this thesis, we have presented two query processing algorithms:

1. r-neighbor range query processing using only the compressed trie. [Algorithm 3.5]

2. a hybrid r-neighbor range query processing using both the compressed trie and MIH. [Algorithm 3.6]

We have proven that Algorithm 3.5 generates **no** null r-variations, so that every lookup guarantees to produce some query result. However, the compressed trie lookup requires extensive tree traversal which is several times more costly than hash table lookup. Algorithm 3.6 is a hybrid approach that utilizes the compressed trie to identify the important bits to flip, but still relies upon MIH for the eventual lookup. This means that we benefit from the strengths of both index structures. The result is that we can reduce the number of hash lookups with minimal amount of tree traversal. One can show that Algorithm 3.6 still generates null r-variations, but far fewer of them.

5.2 Performance Evaluation

We have implemented and evaluated the algorithms using the C programming language. The compressed trie and MIH are both maintained as main memory data structures. In order to thoroughly evaluate the performance characteristics of the algorithms, we have generated synthetic datasets (up to 1 million vectors) and query workload (up to 1000 queries). Our experiments compares the performances of:

1. linear search;

2. naïve MIH; and

3. our approach.

Our experiments show that Algorithm 3.6 reduces the number of hash lookups by orders of magnitude [Figure 4.1, 4.2], while incurring minimal overhead. We also observe that the overall performance is improved by a factor of 2 at times. The performance gain is the most significant when the dataset is sparse, namely the dimensionality N of the binary vectors (actually sub-vectors stored in different indexes for MIH) is such that $|D| \lll 2^N$. This is typically the case in practice (with binary vectors of length > 100). Even at the extreme case when $|D| \simeq 2^N$ (i.e., the dataset is dense), we see that our approach is very comparable to the naïve MIH (Figure 4.2) due to the minimal overhead of the compressed trie.

5.3 Contribution

The contributions of this thesis work can be summarized as follow:

Scientific contribution:

1. A trie-based algorithm for solving the r-neighbors search problem; and
2. A hybrid trie+hash algorithm for solving the r-neighbors search problem.

System engineering contribution: Appendix A provides the implementation details and optimization techniques used to produce fast implementations of the proposed algorithm. We did not invent these implementation tricks. Still putting them together to speedup our proposed algorithm is an engineering contribution.

5.4 Future Work

There are a number of extensions to this thesis we would like to explore in our future research.

1. Nearest neighbors queries

   This thesis is limited to the range query with radius r. A closely related query is that of the k-nearest neighbor (NN) queries. To answer the k-NN queries using hash based index structure, one must resort to the naïve approach of successively expanding the range r until at least k neighbors have been discovered. With the augmentation of a trie, we believe that it's possible to speed up the k-NN queries by utilizing the added local *searchability*.

2. Disk based index and query processing

   Currently all index structures exist in main memory, thus the dataset is limited by the size of the physical memory available to the algorithm. We would like to investigate the issues and solutions of designing disk based index structures. We foresee that there are some interesting issues associated with merging the MIH and compressed trie data structure with well known disk based hashing using B+ tree.

Bibliography

[1] A. Alahi, R. Ortiz, and P. Vandergheynst. Freak: Fast retina keypoint. In *Proceedings of the IEEE Conference on Computer Vision and Pattern Recognition (CVPR)*, pages 510–517, Providence, RI, USA, June 2012.

[2] A. N. Arslan. Efficient approximate dictionary look-up over small alphabets. Technical report, University of Vermont, 2005.

[3] A. N. Arslan. Efficient approximate dictionary look-up for long words over small alphabets. In *Proceedings of the 7th Latin American Symposium on Theoretical Informatics (LATIN)*, volume 3887 of *Lecture Notes in Computer Science*, pages 118–129. Springer Berlin Heidelberg, Valdivia, Chile, March 2006.

[4] A. N. Arslan and O. Egecioglu. Dictionary look-up within small edit distance. In *Proceedings of the 8th Annual International Computing and Combinatorics Conference (COCOON)*, pages 127–136, Singapore, August 2002.

[5] A. Bergamo, L. Torresani, and A. W. Fitzgibbon. Picodes: Learning a compact code for novel-category recognition. In *Proceedings of the Conference on Neural Information Processing Systems (NIPS)*, pages 2088–2096, December 2011.

[6] S. Brin. Near neighbor search in large metric spaces. In *Proceedings of the 21th International Conference on Very Large Data Bases (VLDB)*, Berlin, Germany, September 1995.

[7] G. Brodal and L. Gasieniec. Approximate dictionary queries. In *Combinatorial Pattern Matching*, volume 1075 of *Lecture Notes in Computer Science*, pages 65–74. Springer Berlin Heidelberg, 1996.

[8] M. Calonder, V. Lepetit, C. Strecha, and P. Fua. Brief: Binary robust independent elementary features. In K. Daniilidis, P. Maragos, and N. Paragios, editors, *European Conference on Computer Vision (ECCV)*, volume 6314 of *Lecture Notes in Computer Science*, pages 778–792. Springer Berlin Heidelberg, September 2010.

[9] E. G. Coffman, Jr. and J. Eve. File structures using hashing functions. *Communications of the ACM*, 13(7):427–432, July 1970.

[10] N. G. de Bruijn. A Combinatorial Problem. *Koninklijke Nederlandsche Akademie Van Wetenschappen*, 49(6):758–764, June 1946.

[11] R. De La Briandais. File searching using variable length keys. In *Proceedings of the IRE-AIEE-ACM Western Joint Computer Conference*, pages 295–298, San Francisco, California, March 1959.

[12] M.M. Esmaeili, M. Fatourechi, and R.K. Ward. A robust and fast video copy detection system using content-based fingerprinting. *IEEE Transactions on Information Forensics and Security*, 6(1):213–226, 2011.

[13] M.M. Esmaeili, R.K. Ward, and M. Fatourechi. A fast approximate nearest neighbor search algorithm in the hamming space. *IEEE Transactions on Pattern Analysis and Machine Intelligence (TPAMI)*, 34(12):2481–2488, 2012.

[14] R. Fisher. *General-purpose SIMD with a register: parallel processing on consumer microprocessors.* PhD thesis, School of electrical and computer engineering. Purdue University, 1997.

[15] D. R. Flower. On the properties of bit string-based measures of chemical similarity. *Journal of Chemical Information and Computer Sciences*, 38:379–386, 1998.

[16] E. Fredkin. Trie memory. *Communications of the ACM*, 3(9):490–499, 1960.

[17] J. Gan, J. Feng, Q. Fang, and W. Ng. Locality-sensitive hashing scheme based on dynamic collision counting. In *Proceedings of the 2012 ACM International Conference on Management of Data (SIGMOD)*, pages 541–552, Scottsdale, Arizona, USA, May 2012.

[18] D. Greene, M. Parnas, and F. Yao. Multi-index hashing for information retrieval. In *Proceedings of the 35th Annual Symposium on Foundations of Computer Science (FOCS)*, pages 722–731, Santa Fe, NM, USA, November 1994.

[19] C.A. Healy, R.D. Arnold, F. Mueller, D.B. Whalley, and M.G. Harmon. Bounding pipeline and instruction cache performance. *IEEE Transactions on Computers*, 48(1):53–70, January 1999.

[20] P. Indyk and R. Motwani. Approximate nearest neighbors: Towards removing the curse of dimensionality. In *Proceedings of the T13th Annual ACM Symposium on Theory of Computing (STOC)*, pages 604–613, Dallas, TX, USA, May 1998.

[21] D. E. Knuth. *The Art of Computer Programming, Volume 4, Fascicle 0: Introduction to Combinatorial Algorithms and Boolean Functions (Art of Computer Programming)*. Addison-Wesley Professional, 1 edition, 2008.

[22] D. Kuettel, M. Guillaumin, and V. Ferrari. Segmentation propagation in imagenet. In *European Conference of Computer Vision 2012 (EECV)*, volume 7578 of *Lecture Notes in Computer Science*, pages 459–473. Springer Berlin Heidelberg, October 2012.

[23] J. Landré and F. Truchetet. Fast image retrieval using hierarchical binary signatures. In *Proceedings of the 9th International Symposium on Signal Processing and Its Applications (ISSPA)*, pages 1–4, Sharjah, United Arab Emirates, February 2007.

[24] J. Landré and F. Truchetet. Image retrieval with binary hamming distance. In *Proceedings of the 2nd International Conference on Computer Vision Theory and Applications (VISAPP)*, Barcelona, Spain, March 2007.

[25] C. E. Leiserson, H. Prokop, and K. H. Randall. Using de bruijn sequences to index a 1 in a computer word. *MIT Laboratory for Computer Science*, 1998.

[26] A.X. Liu, Ke Shen, and E. Torng. Large scale hamming distance query processing. In *IEEE 27th International Conference on Data Engineering (ICDE)*, pages 553–564, Hannover, NI, Germany, April 2011.

[27] M. G. MaaB and J. Nowak. Text indexing with errors. Technical report, Institut für Informatik, Technische Universit ät München, 2005.

[28] G. S. Manku, A. Jain, and A. Das Sarma. Detecting near-duplicates for web crawling. In *Proceedings of the 16th International Conference on World Wide Web (WWW)*, pages 141–150, Banff, AB, Canada, May 2007.

[29] M.L. Miller, M.A. Rodriguez, and Ingemar J. Cox. Audio fingerprinting: nearest neighbor search in high dimensional binary spaces. In *IEEE Workshop on Multimedia Signal Processing (MMSP)*, pages 182–185, St. Thomas, VI, USA, December 2002.

[30] F. Mueller and D. B. Whalley. Avoiding conditional branches by code replication. In *Proceedings of the ACM Conference on Programming Language Design and Implementation (PLDI)*, pages 56–66, New York, NY, USA, 1995. ACM.

[31] M. Muja and D. G. Lowe. Fast matching of binary features. In *Proceedings of the 2012 Ninth Conference on Computer and Robot Vision (CRV)*, pages 404–410, Toronto, ON, Canada, May 2012.

[32] M. Norouzi, A. Punjani, and D.J. Fleet. Fast search in hamming space with multi-index hashing. *IEEE Transaction of Pattern Analysis and Machine Inteligence (TPAMI)*, (6), 2014.

[33] J. Oostveen, T. Kalker, and J. Haitsma. Feature extraction and a database strategy for video fingerprinting. In *Proceedings of the 5th International Conference on Recent Advances in Visual Information Systems (VISUAL)*, pages 117–128, Hsin Chu, Taiwan, March 2002.

[34] E. Rublee, V. Rabaud, K. Konolige, and G. Bradski. Orb: An efficient alternative to sift or surf. In *Proceedings of the IEEE International Conference on Computer Vision (ICCV)*, pages 2564–2571, Barcelona, Spain, November 2011.

[35] G. Shakhnarovich. *Learning Task-Specific Similarity*. PhD thesis, MIT, 2006.

[36] G. Shakhnarovich, P. Viola, and T. Darrell. Fast pose estimation with parameter-sensitive hashing. In *Proceedings of the 9th IEEE International Conference on Computer Vision (ICCV)*, volume 2, pages 750–757, Nice, France, October 2003.

[37] M. Theobald, J. Siddharth, and A. Paepcke. Spotsigs: robust and efficient near duplicate detection in large web collections. In *Proceedings of the 31st annual international ACM SIGIR conference on Research and development in information retrieval (SIGIR)*, pages 563–570, Singapore, July 2008.

[38] A. Torralba, R. Fergus, and Y. Weiss. Small codes and large image databases for recognition. In *Proceedings of the IEEE Conference on Computer Vision and Pattern Recognition (CVPR)*, pages 1–8, Anchorage, AK, USA, June 2008.

[39] Y. Weiss, A. Torralba, and R. Fergus. Spectral hashing. In *Proceedings of the Conference on Neural Information Processing Systems (NIPS)*, pages 1753–1760, Vancouver, BC, Canada, December 2008.

[40] Q. Xiao, M. Suzuki, and K. Kita. Fast hamming space search for audio fingerprinting systems. In *Proceedings of the 2011 Conference of The International Society for Music Information Retrieval (ISMIR)*, pages 133–138, Miami, FL, USA, October 2011.

[41] Hong Y. and Yiding W. A lbp-based face recognition method with hamming distance constraint. In *Proceedings of the 4th International Conference on Image and Graphics (ICIG)*, pages 645–649, Chengdu, Sichuan, China, August 2007.

[42] A. C. Yao and F. F. Yao. Dictionary look-up with one error. *Journal of Algorithms*, 25(1):194 – 202, 1997.

[43] X. Zhang, J. Qin, W. Wang, Y. Sun, and J. Lu. Hmsearch: An efficient hamming distance query processing algorithm. In *Proceedings of the 25th International Conference on Scientific and Statistical Database Management (SSDBM)*, pages 19:1–19:12, Baltimore, Maryland, July 2013.

Appendix

Appendix A

Implementation Details

In this section we provide technical details that are needed for achieving the performance showed in Chapter 4. These details are not our original contribution, but their importance cannot be underestimated. The first type of optimization is oriented to the instruction set generated by the compiler (see Section A.1). The second type of optimization is regarding the optimum implementation of the bit manipulations functions (see Section A.2).

A.1 Pipeline and Instruction Cache Performance

A.1.1 Trie Implementation Using Array to Increase Instruction Cache Performance

The most common way to implement a Tree (remember that a trie is also a tree) is to use pointers. However, the use of pointers is not the most efficient way to implement these. Accessing memory through pointers is expensive because of CPU level caching.

The trie (and trees in general) can be more efficiently implemented using a static array. Using this implementation, pointers to children are replaced by array offsets. Using the array offsets has better CPU cache hit rate and with that a better memory access throughput.

A.1.2 Avoiding Instruction Branching on Left-Right Trie Decision When Traversing the Trie

The conditional jump machine language instructions (aka branches), may be generated by many statements, among which there are the if-else. Modern processors handle branches efficiently only if they can predict them. In case of prediction error, the steps already done by the pipeline on the subsequent instructions are useless and the processor must restart from the branch destination instruction [30].

The branch prediction is based on the previous iterations on the same instruction. If the branches follow a regular pattern, the predictions are successful. The case where the branch instruction has a random outcome results in the prediction being on average correct half of the times. However random distribution means that actual results will vary from being always right to always wrong in a Gaussian shaped distribution [19].

Branching is heavily used while traversing a Trie since at a given node we will move to the left or right child according to the value of the bit at the *splitBit* position (see Section 3.2). The following pseudocode shows this common structure found in all Trie algorithm:

```
if $q[splitPos] = 0$ then
    goto $node.leftChild$
else
    goto $node.rightChild$
```

Unfortunately, the nature of the Trie makes branches hard to predict. It is better to replace bad predictions with a slow sequence of instructions that may result in a speed up [30]. To avoid the branching at instruction level, all nodes instead of having the field $leftChild$ and $rightChild$, have a length 2 array called *children*, this way the branching can be removing by accessing the child according to the value at the *splitBit* position by just doing:

```
goto $node.children[q[splitPos]]$
```

A.2 Bit Manipulation Functions

One essential part in the implementation of the proposed algorithm is the bit manipulation functions since they are the core in all bitwise trie functions. Here we provide the details of how different bit manipulation functions can be implemented.

A.2.1 Accessing Bit Positions

In the pseudocode of the different functions, accessing the bit at position p of vector $\mathbf{x}$ have been represented as $\mathbf{x}[p]$. However in practice binary vectors are represented in a compact form in machine words, so you can not access bits as in an array.

Algorithm A.1 shows how to test for a bit at a given position.

Algorithm A.1 testBit algorithm

```
procedure TESTBIT(x, p)
    return $(\mathbf{x} \gg p) \wedge 1$
```

Similarly, Algorithm A.2 shows how to change the value of the bit at a given position.

Algorithm A.2 flipBit algorithm

```
procedure TESTBIT(x, p)
    return $\mathbf{x} \wedge (1 \ll p)$
```

A.2.2 bitScanForward

The function *bitScanForward* plays a fundamental role in the compressed trie insertion algorithm. This routine is used to find the index of the least significant 1 bit.

This function can be implemented efficiently based on the principle that a multiplication with a power of two value acts like a left shift by it's exponent [21]. If $\mathbf{x}$ is non-zero, $\mathbf{x}\ \&\ -\mathbf{x}$ turns all bits to zero except the least significant 1. This operation is called isolating the least significant 1.

With the isolated the least significant 1 bit, the vector x can only have $|\mathbf{x}|$ possibles values: $2^0, 2^1, \ldots 2^{|\mathbf{x}|}$. Those values generates a unique number between 0 and $|\mathbf{x}|-1$ when multiplied by the a De Brujin sequence of size $|\mathbf{x}|$ [10]. With these unique numbers, a lookup on an array can be performed to get the index of the least significant 1 bit [25]. Algorithm A.3 shows a pseudo-code implementation of the *bitScanForward* function.

Algorithm A.3 bitScanForward algorithm for 64 bits

```
index64[64] ← {0, 1, 48, 2, 57, 49, 28, 3,
               61, 58, 50, 42, 38, 29, 17, 4,
               62, 55, 59, 36, 53, 51, 43, 22,
               45, 39, 33, 30, 24, 18, 12, 5,
               63, 47, 56, 27, 60, 41, 37, 16,
               54, 35, 52, 21, 44, 32, 23, 11,
               46, 26, 40, 15, 34, 20, 31, 10,
               25, 14, 19, 9, 13, 8, 7, 6}

procedure BITSCANFORWARD($\mathbf{x}$)
    $debruijn64 \leftarrow$ 0x03f79d71b4cb0a89
    return $index64[((\mathbf{x} \wedge -\mathbf{x}) * debruijn64) \gg 58]$
```

A.2.3 Hamming Distance and popCount

Testing for the Hamming distance between two binary vectors can be performed by doing an *xor* between them and count the number of ones bits. The function to determine how many one bits exists in the given vector is know in the literature as *popCount*. Recent x86-64 processors (AMD K10 - SSE4a, Intel Nehalem - SSE4.2) provide a built-in 64-bit popCount instruction. However, it is possible to implement a better alternative to the built-in.functino.

One efficient way to implement the *popCount* function is to use the Fis97 technique [14]. This approach deals with counting bits of duos, to aggregate the duo-counts to nibbles and bytes to finally sum all bytes together [21].

A bit-duo (two neighboring bits) can be interpreted with bit $0 = a$, and bit $1 = b$ as

$$duo := 2b + a$$

The duo population is

$$popCount(duo) := b + a$$

which can be archived by

$$(2b + a) - (2b + a)/2$$

The bit-duo has up to four states with population count from zero to two as demonstrated in table A.1.

Table A.1: States for bit-duo popCount

$\mathbf{x}$	$\mathbf{x/2}$	$\mathbf{x - x/2} = popCount(\mathbf{x})$
00	00	00
01	00	01
10	01	01
11	01	10

As can be seen, only the lower bit is needed from $\mathbf{x}/2$. SWAR-wise, one needs to clear all "even" bits of the div 2 subtrahend to perform a subtraction of all duos. For a 64-bit architecture, the mask would be:

$$\texttt{0x5555555555555555} = 101010\ldots10$$

Then, obtaining all duo-count can be done doing:

$$\mathbf{x} = \mathbf{x} - ((\mathbf{x} \gg 1) \wedge \texttt{0x5555555555555555});$$

The next step is to add the duo-counts to populations of four neighboring bits, the nibble-counts, which may range from zero to four. SWAR-wise it is done by masking odd and even duo-counts to add them together. For a 64 bit architecture, the mask would be:

$$\texttt{0x3333333333333333} = 11001100\ldots1100$$

All nibble-counts can be obtained by doing:

$$\mathbf{x} = (\mathbf{x} \wedge \texttt{0x3333333333333333}) + ((\mathbf{x} \gg 2) \wedge \texttt{0x3333333333333333})$$

In the same way it is possible to get the byte-populations from two nibble-populations. The mask for a 64 bit architecture would be:

$$\texttt{0xf0f0f0f0f0f0f0f} = 11110000\ldots11110000$$

Then the byte-wise sum can be obtained by:

$$\mathbf{x} = (\mathbf{x} + (\mathbf{x} \gg 4)) \wedge \texttt{0x0f0f0f0f0f0f0f0f}$$

The last step is to sum all bytes to get the final population count. This can be done by multiplying the vector of byte-counts with the fraction -1/255 to get the final result in the most significant byte [21]. For a 64 bit architecture, this fraction is:

$$-1/255 = \texttt{0x0101010101010101} = 10000000\ldots10000000$$

Since the count is in the most significant byte, it can be shifted right to get the final count. For a 64 bit architecture it would be:

$$\mathbf{x} = (\mathbf{x} * \texttt{0x0101010101010101}) \gg 56$$

Putting all pieces together we get the algorithm A.4.

Algorithm A.4 popCount algorithm for 64 bits

```
procedure popCount($\mathbf{x}$)
    $\mathbf{x} = \mathbf{x} - ((\mathbf{x} \gg 1) \wedge \texttt{0x5555555555555555})$
    $\mathbf{x} = (\mathbf{x} \wedge \texttt{0x3333333333333333}) + ((\mathbf{x} \gg 2) \wedge \texttt{0x3333333333333333})$
    $\mathbf{x} = (\mathbf{x} + (\mathbf{x} \gg 4)) \wedge \texttt{0xf0f0f0f0f0f0f0f}$
    return $(\mathbf{x} * \texttt{0x101010101010101}) \gg 56$
```