Ultra-Fast Similarity Search Using Ternary Content Addressable Memory

Anat Bremler-Barr
The Interdisciplinary Center
Herzliya, Israel
bremler@idc.ac.il

Yotam Harchol
The Hebrew University
Jerusalem, Israel
yotamhc@cs.huji.ac.il

David Hay
The Hebrew University
Jerusalem, Israel
dhay@cs.huji.ac.il

Yacov Hel-Or
The Interdisciplinary Center
Herzliya, Israel
toky@idc.ac.il

ABSTRACT

Similarity search, and specifically the nearest-neighbor search (NN) problem is widely used in many fields of computer science such as machine learning, computer vision and databases. However, in many settings such searches are known to suffer from the notorious *curse of dimensionality*, where running time grows exponentially with d. This causes severe performance degradation when working in high-dimensional spaces. Approximate techniques such as *locality-sensitive hashing* [2] improve the performance of the search, but are still computationally intensive.

In this paper we propose a new way to solve this problem using a special hardware device called *ternary content addressable memory* (TCAM). TCAM is an associative memory, which is a special type of computer memory that is widely used in switches and routers for very high speed search applications. We show that the TCAM computational model can be leveraged and adjusted to solve NN search problems in a single TCAM lookup cycle, and with linear space. This concept does not suffer from the curse of dimensionality and is shown to improve the best known approaches for NN by more than *four orders of magnitude*. Simulation results demonstrate dramatic improvement over the best known approaches for NN, and suggest that TCAM devices may play a critical role in future large-scale databases and cloud applications.

1. INTRODUCTION

Multidimensional nearest neighbor search (NN) lies at the core of many computer science applications. Given a database of objects and a query, we wish to find the object in the database most similar to the query object. Commonly, the objects are mapped to points in high-dimensional metric space. In this context, given a query point $q \in \mathbb{R}^d$ and a set of points $S = \{p_i\}_{i=1}^n$, $p_i \in \mathbb{R}^d$, the goal is to find a point $p \in S$ most similar to the query point q under some distance metric. In addition to the exact NN, many variants of this problem exist. The NN and its variants are utilized in a wide range of applications, such as similarity search, spatial search, object recognition, classification and detection, to name a few [4, 7, 15, 17, 21].

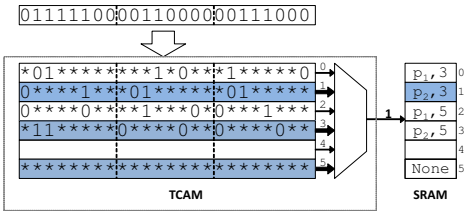

Figure 1: Diagram of the TCAM lookup process. The query is compared to all entries in parallel and the index of the first matching entry is used to find the corresponding point and cube size.

When the dimensionality of the points is small, many solutions were proven to be very effective. These include mainly space partitioning techniques [3, 5, 25]. However, when the number of the data points is large (in the order of tens of thousands or higher) and the dimensionality is high (in the order of tens or hundreds), the exact solutions break down and produce exponential time complexity[1] [2,27]. This problem is widely known as the *curse of dimensionality*. To overcome the curse of dimensionality, *approximated nearest neighbor* (ANN) solutions such as *locality sensitive hashing* (LSH) [2] are commonly used.

In this paper, we present super high-speed algorithms for the NN problem using a special existing hardware device called *ternary content-addressable memory (TCAM)*. The proposed algorithms solve the ANN problem with ℓ_∞-normed distance using *a single TCAM lookup*, and linear space.

TCAM is an associative memory module that is widely used in high-speed networking devices such as switches and routers, mainly for IP lookup and packet classification [13, 22]. TCAM memory is composed of an array of ternary words, each consisting of ternary digits, namely: 0, 1, or *.

[1]Exact brute-force search works in time that is linear to n and d, but is very slow for high n and d. Space partitioning techniques are exponential in d.

DaMoN 2015, June 1, 2015, Melbourne, VIC, Australia.

http://dx.doi.org/10.1145/2771937.2771938.

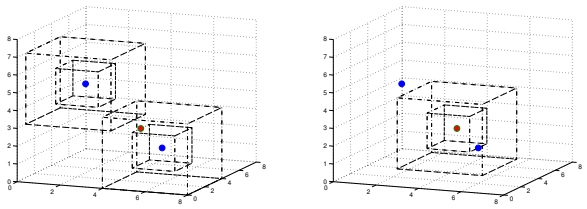

Figure 2: Illustration of the two alternative algorithms for Nearest Neighbor Search using TCAM.

Given a query word, TCAM returns in a single lookup cycle (a few nanoseconds) the location in the memory array that matches the query. When a key matches multiple TCAM entries, the TCAM returns the location of the first matching entry. This process is illustrated in Figure 1. Because the '*' digits serve as 'wild cards' that can be matched with either '0' or '1' we can design a special type of code for encoding either a particular point or a particular range area (cube) in high-dimensional space. This is the key to using TCAM in NN searches.

To the best of our knowledge, using TCAM for nearest neighbor search has been considered only once, by Shinde et al. [26] who proposed the TLSH scheme, where a TCAM device is utilized to implement LSH with a series of TCAM lookup cycles. As shown later, our technique is faster than TLSH and requires less TCAM space. Unlike TLSH, our technique allows hot updates and in not probabilistic.

The general framework of our algorithms is simple. Given a database of n points in the d-dimensional space, we store in the TCAM d-dimensional cubes around these points (each TCAM entry stores a representation of a single cube using a ternary code). For each point, we store cubes in increasing edge lengths. Hence, given a query point, we match it against the TCAM entries; the resulting cube (or more specifically, the corresponding point) is the nearest neighbor under ℓ_∞, while the approximation ratio is determined by the size difference between the nested cubes. This approach is illustrated in Figure 2 (left). Alternatively, one can encode the points precisely and match the TCAM with cubes of increasing sizes, as illustrated in Figure 2 (right). Naturally, such an algorithm has a much smaller memory footprint, but it may require several TCAM queries to obtain a result (instead of a single query in our first algorithm).

We implemented our algorithms and used them for an image similarity search application. We show in Section 5 that our approximated algorithms achieve results that are comparable to the best prior-art approximated solutions, in rates higher by four orders of magnitudes than of other solutions.

2. BACKGROUND

2.1 Problem Definition

The nearest neighbor search problem is formally defined as follows for points in a discrete space of dimension d:

DEFINITION 1. *Given a set of data points $S = \{p_i\}_{i=1}^n$, $p_i \in \mathbb{Z}^d$ and a query point $q \in \mathbb{Z}^d$,* THE NEAREST-NEIGHBOR PROBLEM *is to find the point $s^* = \arg\min_{s \in S} D(s,q)$, where $D(s,q)$ is a distance between s and q.*

The c–APPROXIMATE NEAREST-NEIGHBOR PROBLEM (c-ANN) searches for a point $p \in S$ such that $D(p,q) < c \cdot D(s^*,q)$, where s^* is the nearest point to q.

The THE k–NEAREST-NEIGHBOR PROBLEM finds a set $S' \subseteq S$ of k points such that for each $p' \in S'$ and $p \in S \setminus S'$, $D(p',q) \leq D(p,q)$. We show how to adapt our technique to solve this problem as well in Section 4.2.1.

A simpler problem that we will use as a building block in our algorithms only searches for a neighbor close enough to the query point, or discovers that there is no such neighbor at all:

DEFINITION 2. *Given a set of points $S = \{p_i\}_{i=1}^n$, $p_i \in \mathbb{Z}^d$, and a query point $q \in \mathbb{Z}^d$, let $d^* = \min_{s \in S} D(s,q)$. The* r–NEAR-NEIGHBOR REPORT PROBLEM *is to find the point $s' \in S$ such that $D(s',q) \leq r$ if $d^* \leq r$, and to return* false *if $d^* > r$.*

Note that under ℓ_∞, a solution for the r–NEAR-NEIGHBOR REPORT PROBLEM is a data point within the d-dimensional cube of edge length $2r$ that is centered in the query point. Our framework for solving c-ANN can be viewed as solving (either in parallel or sequentially) a series of r–NEAR-NEIGHBOR REPORT PROBLEM instances for increasing values of r. As pointed out in [12], this solves the c-APPROXIMATE NEAREST NEIGHBOR PROBLEM, where the approximation ratio is determined by the maximum ratio between consecutive values of r.

2.2 Ternary Content Addressable Memory (TCAM)

Contemporary TCAM devices operate at very high rates, between hundreds of millions to more than one billion queries per second [11,23]. These devices have about 20-80 megabits of memory that can be configured to accommodate entries of up to about 640-bits wide (the wider the entry, the fewer entries can be stored on the chip).

The downsides of TCAM are that it is power hungry and relatively expensive, compared to a standard DRAM chip. A high-density TCAM consumes 12 to 15W per chip when the entire memory is used [22]. However, compared to compute units and coprocessors such as CPU, GPU, or FPGAs, TCAMs' power requirement, heating and price are actually lower, and become similar only when connecting multiple TCAMs in parallel, as usually done in high end networking equipment. For example, Intel's E7-4870 CPU consumes 130W [8], and Nvidia's Tesla K80 GPU consumes up to 300W [18]. Another downside could be that currently, a TCAM cannot be easily deployed on a standard PC, as they are manufactured for networking equipment.

However, due to their impressive adoption for networking devices, TCAMs are becoming larger, faster, less power hungry and less expensive. We speculate that this trend will continue. Inspired by the adoption of Graphics Processing Units (GPUs) for general purpose parallel computing in recent years, we suggest that TCAMs may be useful for other tasks outside the networking field.

3. ENCODING SCHEME FOR INTERVALS

Our goal is to encode a cube of arbitrary edge length around a point, in a way that provides a solution (with a single TCAM query) for the r-NEAR NEIGHBOR REPORT PROBLEM, and therefore also for the c-APPROXIMATE NEAREST NEIGHBOR PROBLEM. The basic intuition can be seen in Figure 2: We would like to encode cubes and points such that using the TCAM we could find whether a cube contains a given point, or to find the smallest cube that contains a given point.

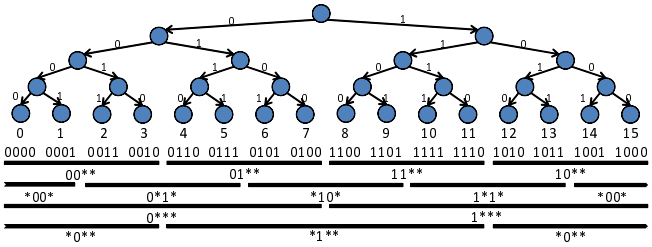

Figure 3: BRGC encoding tree for points in range $[0, 16)$, and ternary representations of intervals of lengths $4, 8$ that can be represented using this encoding.

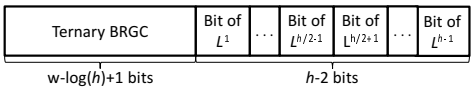

Figure 4: Encoding structure for a point or interval of length h

We represent each such cube with a *single TCAM entry*. We assume that edge lengths are bounded by some constant $h_{\max}$ (namely, large cubes are not interesting as they provide very little information).

3.1 Basic Definitions

We begin with some basic definitions that will be used throughout the rest of this section. First, we define a ternary bit-wise comparison:

DEFINITION 3. *Let $a = a_0, \ldots, a_m$ and $b = b_0, \ldots, b_m$ be two ternary words ($a_i, b_i \in \{0, 1, *\}$). a* **matches** *b, denoted $a \approx b$, if and only if for every $i \in \{0, \ldots, m\}$, either $a_i = b_i$, or a_i is $*$, or b_i is $*$.*

We begin by defining an encoding function `tcode` for points and intervals. Let $\mathcal{U} = [0, 2^w) \subset \mathbb{N}_0$ be an interval on the natural line. Our encoding function `tcode` encodes either a point $p \in \mathcal{U}$, or an interval $I \subseteq \mathcal{U}$. It is important to note that we treat $\mathcal{U}$ as a cyclic 'wrap-around' space and thus throughout this paper, any interval $[x, y)$ refers in fact to $[x, y \bmod 2^w)$.

The result of the encoding function is either a binary word (for points) or a ternary word (for intervals), and we expect that a ternary match $\texttt{tcode}(p) \approx \texttt{tcode}(I)$ will imply that the point p is inside the interval I. This is formally defined as follows:

DEFINITION 4. *An encoding function* `tcode` *is* **admissible** *if for every point $p \in \mathcal{U}$ and every interval $I \subseteq \mathcal{U}$, $\texttt{tcode}(p) \approx \texttt{tcode}(I)$ if and only if $p \in I$. Furthermore, for any point $p \in \mathcal{U}$, $\texttt{tcode}(p)$ does not contain '$*$' symbols.*

3.2 Binary-Reflected Gray Code for TCAM

The *binary-reflected Gray code* (BRGC) [10] is a binary encoding of integers in a contiguous range such that the codes of any two consecutive numbers differ by a single bit. A b-bits BRGC is constructed recursively by reflecting a $(b-1)$-bits BRGC[2]. An example for values in $[0, 16)$ is shown in Figure 3.

[2]The BRGC of a value x can be directly calculated using the following formula: $x \oplus (x >> 1)$, where $\oplus$ and $>>$ are the bitwise operations of XOR and Right Shift, respectively.

By wildcarding some of the bits of a BRGC codeword we can create a ternary interval representation. For example, as can be seen in Figure 3, the ternary word `01**` matches all values in the interval $[4, 7]$, and the ternary word `*1**` matches all values in the interval $[4, 11]$. In fact, when looking at this tree representation of the BRGC encoding, we observe that all intervals that exactly contain a full sub-tree, or two adjacent full sub-trees, can be represented using a single *ternary* BRGC codeword (namely, a BRGC codeword where some of the 0-1 bits were replaced by '$*$' symbols).

Before formulating and proving[3] this observation we define the following terms that will be used in the proof: A *k-prefix* is a ternary word in which the k least significant bits are '$*$' and the rest are either 0 or 1. A *k-semi-prefix* is a ternary word in which the k least significant bits are '$*$', one additional bit is also '$*$', and the rest are either 0 or 1.

THEOREM 1. *If all values are BRGC-encoded, then a single ternary BRGC codeword suffices to admissibly encode an interval $I = [x, y \bmod 2^w)$ if and only if there exist non-negative integers i, k, for which $x = i \cdot 2^k$ and $y = (i+2) \cdot 2^k$. Specifically, one of the following cases holds:*

1. *If i is even, the ternary codeword is a $(k+1)$-prefix.*
2. *If i is odd, the ternary codeword is a k-semi-prefix.*

Theorem 1 implies that when encoding ranges of length h, the $\log_2(h) - 1$ least significant bits are always '$*$' thus one can save TCAM space by omitting these uninformative bits.

3.3 An Encoding Function for Intervals

Ternary BRGC codewords cannot encode all intervals but only those satisfying Theorem 1. We now present an encoding scheme for any interval up to some maximal length h_{max}, which is based on the observation of Theorem 1.

We call those intervals that can be encoded using a single ternary BRGC codeword *trivial intervals*, and all other intervals *nontrivial intervals*. For example, $[4, 7]$ is a trivial interval, but $[5, 8]$ is not. We use this as a building block for an encoding scheme that encodes in a single ternary word trivial and nontrivial intervals. We append *extra bits* to the end of BRGC codewords, as depicted in Figure 4: $w - \log_2(h) + 1$ bits are used for the binary BRGC encoding of a point $p \in \mathcal{U}$, or for the ternary BRGC encoding of some trivial interval I. To encode nontrivial intervals of length $h = 2^k$ $(k \in \mathbb{N}_0)$, at most $h - 2$ bits are added as extra bits.

The key idea of our scheme is to divide all intervals of some length $h = 2^k$ $(k \in \mathbb{N}_0)$ into h *layers*, such that a layer L_h^i is the set of all intervals $[x, x+h)$ for which $x \bmod h = i$. Note that two of these layers contain only trivial intervals (L_h^0 and $L_h^{2^{k-1}}$). We are left with $h - 2$ layers that contain nontrivial intervals. We assign an extra bit for each layer of nontrivial intervals. The value of this bit alternates between adjacent intervals in the same layer, such that for any pair of consecutive intervals in the same layer, the value of the bit corresponding to this layer is different. Hence, for a point $p \in \mathcal{U}$, $\texttt{tcode}(p)$ is the $1 + w - log_2(h)$ most significant bits of $BRGC(p)$, concatenated with $h - 2$ extra bits. The value of the i^{th} extra bit corresponds to layer L_h^i and is set to $\left\lfloor \frac{p-i}{h} \right\rfloor \bmod 2$.

[3]Proofs for all theorems and lemmas in the paper are in Appendix.

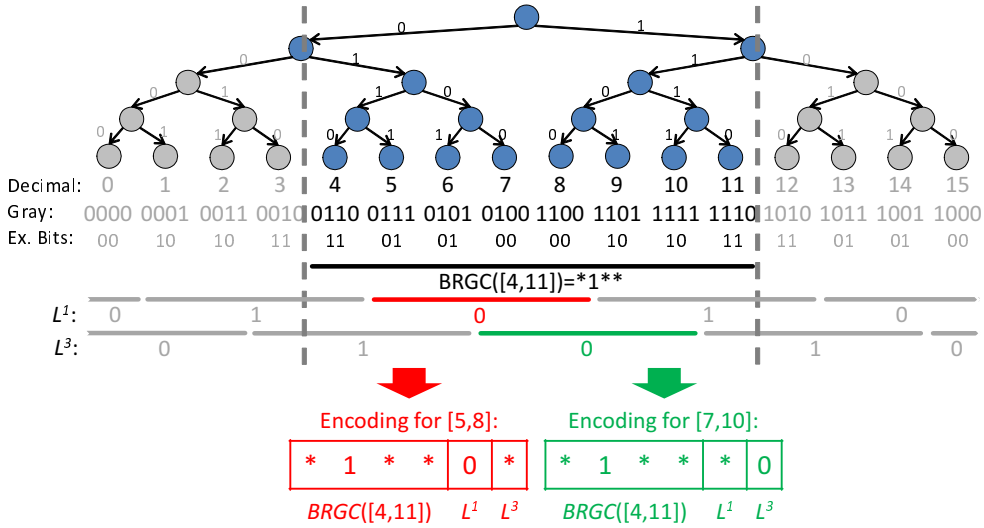

Figure 5: Example of the encoding structure of two nontrivial intervals.

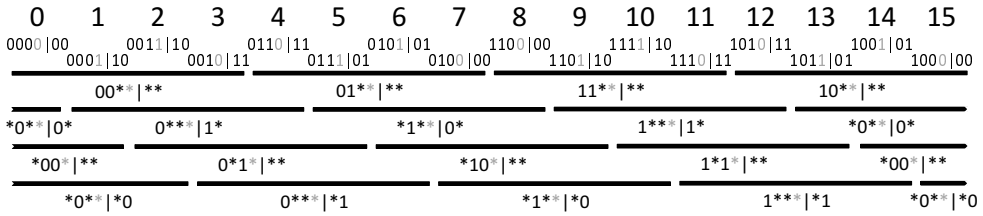

Figure 6: Encoding for all intervals of length 4 and points in range $[0, 16)$. Left bits are the ternary BRGC encoding. Right bits are the extra bits for nontrivial layers. The bits in gray can be removed as explained in Section 3.2

For nontrivial intervals we define their *cover interval* as the smallest trivial interval that contain them. For example, the cover interval of the nontrivial interval $[5, 8]$ is $[4, 11]$. Formally, the cover interval is defined as follows:

DEFINITION 5. *For any nontrivial interval of length $h = 2^k$ ($k \in \mathbb{N}_0$), $I = [x, x + h)$, let the* **cover interval** *of I, denoted by $\texttt{cover}(I)$, be the interval $[\lfloor x/h \rfloor \cdot h, (\lfloor x/h \rfloor + 2) \cdot h)$.*

The main property of cover intervals is formally given in the following lemma:

LEMMA 1. *For any interval $I = [x, x+h)$ of length $h = 2^k$ ($k \in \mathbb{N}_0$), $\texttt{cover}(I)$ fully contains I.*

The cover interval $\texttt{cover}(I)$ helps us distinguish I from other intervals in the same layer. For interval $I = [x, x+h) \in L_h^i$ of length $h = 2^k$ ($k \in \mathbb{N}_0$), $\texttt{tcode}(I)$ starts with the $1 + w - log_2(h)$ most significant bits of the ternary BRGC representation of I, if I is trivial, or of $\texttt{cover}(I)$, if I is nontrivial. Then, $h - 2$ extra bits are concatenated one after the other where the i^{th} bit is either $*$ if $I \notin L_h^i$ or $\left\lfloor \frac{x-i}{h} \right\rfloor$ mod 2 otherwise (namely, all the extra bits except one are $*$). An example of this encoding is shown in Figure 5 for two nontrivial intervals.

Our main result is that our encoding function, $\texttt{tcode}$, is an *admissible encoding function* for intervals of any length $h = 2^k$ ($k \in \mathbb{N}_0$). The total length of the admissible encoding produced by $\texttt{tcode}$ for a single point or interval is hence $w - \log(h) + h - 1$.

Before proving this result (Theorem 2), we introduce the following two technical lemmas:

LEMMA 2. *If an interval I is nontrivial, then no other interval from the same layer is fully contained in $\texttt{cover}(I)$.*

Based on this property of cover intervals, we can completely distinguish between intervals in the same layer using the extra bits we added to the ternary BRGC encoding:

LEMMA 3. *Let $I = [x, x+h)$, where $h = 2^k$, be an interval in L_h^i. For every point p in $\texttt{cover}(I)$, if $p \in I$, p has the same bit value as I, and if $p \notin I$, then p has the opposite bit value.*

We now turn to the main theorem.

THEOREM 2. *The function $\texttt{tcode}$ is an admissible encoding function for ranges of length $h = 2^k$.*

Figure 6 shows the encoding of all intervals of length 4 in range $[0, 16)$. Note that the first and third layers do not require extra bits, so these are both set to $*$ in their encoding. In other layers, the corresponding extra bit alternates between intervals in the same layer. For example, the interval $[1, 4]$, which cannot be encoded solely using a ternary BRGC codeword, is encoded as $\texttt{0***1*}$, where the fifth bit is the extra bit that corresponds to the second layer. Only points in $[1, 4]$ match this encoding.

Algorithm 1 Encoding Function for a Point p

```
1: function tcode (p, h_max)
2:                          ▷ p - point, h_max - maximal interval length
3:     word ← BRGC(p) >> (log_2(h_max) − 1)   ▷ Bitwise shift
4:     for i ← 0 to (h_max − 1) do
5:         if layer is skipped then
6:             continue                ▷ Optional - encode less layers
7:         end if
8:         if layer is nontrivial (i ≠ 0 and i ≠ h_max/2) then
9:             b ← ⌊(p − i)/h_max⌋ mod 2
10:            word ← (word << 1)|b                ▷ Bitwise OR
11:        end if
12:    end for
13:    return word
14: end function
```

3.4 Encoding Multiple Interval Lengths

Given our encoding function for intervals of some maximal length h_{max} we can encode, without using more bits, *all intervals whose lengths are smaller than* h_{max} as well. We define a logical conjunction operation, denoted by $\sqcap$, to encode the intersection of two intervals.

Given two ternary digits a and b, we define $a \sqcap b$ as follows: If a or b are $*$ then $a \sqcap b = *$. If $a = b$ then $a \sqcap b = a = b$. Otherwise, $a \sqcap b = \perp$, which means an undefined output, and we later make sure to never get such an output when using this operation. For two ternary words, $a = a_0, \ldots, a_m$ and $b = b_0, \ldots, b_m$, the conjunction $c = a \sqcap b$ is the ternary word where $c_i = a_i \sqcap b_i$. If at least one of the symbols c_i is $\perp$, then c is also marked as $\perp$ and is not defined. Note that the conjunction operation is independent of the specific encoding function.

The essence of the conjunction operation is captured in the following two lemmas:

LEMMA 4. *For any point $p \in \mathcal{U}$ and any two intervals $I_1, I_2 \subseteq \mathcal{U}$, if* `tcode` *is an admissible encoding function for I_1 and I_2, then* $\texttt{tcode}(p) \approx \texttt{tcode}(I_1) \sqcap \texttt{tcode}(I_2)$ *if and only if $p \in I_1 \cap I_2$.*

LEMMA 5. *If* `tcode` *is an admissible encoding function for I_1 and I_2, and the result of* $\texttt{tcode}(I_1) \sqcap \texttt{tcode}(I_2)$ *is $\perp$ then $I_1 \cap I_2 = \emptyset$.*

We assume that there is a value $h_{max} = 2^{k_{max}}$, which is the maximum length we should consider. Any interval $[x, x+h)$ of length $h < h_{max}$ (h is not necessarily a power of 2 anymore) can be written as the intersection of two intervals of length h_{max} as follows: $[x, x+h) = [x+h-h_{max}, x+h) \cap [x, x+h_{max})$.

Using the conjunction operator and Lemma 4 we can construct the code for intervals of any length $h \leqslant h_{max}$, with $h_{max} - 2$ extra-bits: $\texttt{tcode}([x, x+h)) = \texttt{tcode}([x+h-h_{max}, x+h)) \sqcap \texttt{tcode}([x, x+h_{max}))$. We also know from Lemma 5 that $\texttt{tcode}([x, x+h))$ is not $\perp$ as the intersection is never empty.

The encoding function $\texttt{tcode}(p)$ for some point p when using any range lengths up to h_{max} is shown in Algorithm 1. When encoding an interval $I = [s, t]$ of length $h \leqslant h_{max}$, Algorithm 2 is used to obtain $\texttt{tcode}(I)$.

The length of the resulting encoding of a point $p \in \mathcal{U}$ or an interval $I \subseteq \mathcal{U}$ is therefore $w - \log_2(h_{max}) + h_{max} - 1$.

3.5 Encoding Multiple Dimensions

So far, we only encoded points and intervals in a single dimension. However, the expansion to d dimensions is trivial: to encode a d-dimensional point, or a d-dimensional cube, each coordinate is encoded using the `tcode` function, and the codewords of all d coordinates are concatenated into a single ternary word. Algorithm 3 and Algorithm 4 show the encoding process for a d-dimensional point p, and for a d-dimensional cube centered at some point p, respectively. The length of the codewords is hence $d \cdot (log(w/h_{max}) + h_{max} - 1)$.

Algorithm 2 Encoding Function for an Interval $[s, t]$

```
1: function tcode ([s, t], h_max)
2:                  ▷ [s, t] - interval, h_max - maximal interval length
3:     if t − s + 1 ≠ h_max then
4:                          ▷ Encode interval as an intersection
5:         I_1 = [s, s + h_max − 1]
6:         I_2 = [t − h_max + 1, t]
7:         Γ ← {I_1, I_2}
8:     else                                  ▷ Encode range directly
9:         Γ ← {[s, t]}
10:    end if
11:    result ← 0
12:    count ← 0
13:    for [x, y] ∈ Γ do
14:        mask ← 0
15:        for i ← x + 1 to y do
16:            mask ← mask|(BRGC(i − 1) ⊕ BRGC(i))
17:                                  ▷ bitwise OR and XOR
18:        end for
19:        word ← BRGC(x) >> (log_2(h_max) − 1)
20:        mask ← mask >> (log_2(h_max) − 1)
21:        for i ← 0 to (h_max − 1) do
22:            if layer is skipped then
23:                continue          ▷ Optional - encode less layers
24:            end if
25:            if layer is nontrivial (i ≠ 0 and i ≠ h_max/2) then
26:                if x mod h_max ≠ i then          ▷ Irrelevant layer
27:                    mask ← (mask << 1)|1              ▷ Put a '*'
28:                    word ← word << 1
29:                else                          ▷ [x, y] is in this layer
30:                    mask ← mask << 1
31:                    b ← ⌊(x − i)/h_max⌋ mod 2
32:                    word ← (word << 1)|b
33:                end if
34:            end if
35:        end for
36:        if count > 0 then
37:            result ← result ⊓ (word, mask)
38:        else
39:            result ← (word, mask)
40:        end if
41:        count ← count + 1
42:    end for
43:    return result
44: end function
```

4. NEAREST NEIGHBOR SEARCH ON TCAM

We use our encoding function to implement applications that use the nearest-neighbor problem or its variants, on TCAM. We show several such variants in this section, and by experiments and simulations we show that our technique can improve their performance by orders of magnitudes.

4.1 Approximate Nearest-Neighbor Search

Our APPROXIMATE NEAREST-NEIGHBOR SEARCH algorithms solve in fact multiple instances of the r-NEAR NEIGHBOR REPORT PROBLEM for increasing values of r. In ℓ_∞, the

Algorithm 3 Encoding Function for a d-Dimensional Point

```
1: function TCODE(p[], d, h_max)
2:     word ← ε
3:     for i ← 1 to d do
4:         word ← word + tcode(p[i], h_max)
5:     end for
6:     return word
7: end function
```

Algorithm 4 Encoding Function for a d-Dimensional Cube

```
1: function TCODE(p[], d, h, h_max)
2:     word ← ε
3:     for i ← 1 to d do
4:         word ← word+tcode([p[i]−⌊h/2⌋, p[i]+⌊h/2⌋], h_max)
5:     end for
6:     return word
7: end function
```

value of r defines a cube around each data point p such that for all query points q inside that cube, p is a valid solution of the r-Near Neighbor Report Problem with q, and for all query points outside that cube p is not a valid solution.

Our *time-efficient method* solves the Approximate Nearest-Neighbor Search in ***a single TCAM lookup***. Given a set $\mathcal{H}$ of edge lengths, let $h_{max} = \max_h \mathcal{H}$. For each point $p \in S$ and $h \in \mathcal{H}$ we store a TCAM entry representing a d-dimensional cube centered at p, with edge length h (see Algorithm 4). Entries are sorted by the value of h: the smaller h is, the higher the priority of the entry.

Given a query point $q \in [0, w)^d$, we use `tcode` to build a d-dimensional point representation for maximal edge length of h_{max} (see Algorithm 3), and use a *single TCAM lookup* to find the smallest cube that contains the point q. The TCAM returns the highest priority entry that matches, which is the entry of the cube that is centered at some point p, has the shortest edge length, and contains q. An example is shown in Figure 2 (left).Note that in general, p is not necessarily the exact nearest neighbor of q (as there may be more than one such cube with the same edge length h). However, the distance (under ℓ_∞) of q from its exact nearest neighbor is strictly more than $\left\lfloor \frac{1}{2} \max_{h' \in \mathcal{H}} \{h' < h\} \right\rfloor$. As we will show later, by carefully choosing the edge length set, we can obtain a $c = 1 + \varepsilon$ approximation factor, where the size of $\mathcal{H}$ is inversely proportional to ε.

In our *memory-efficient method*, the data points and query points switch roles: we store in the TCAM a single entry for each data point, as obtained by Algorithm 3. The order of the entries does not matter. Upon a query q, we construct a sequence of $|\mathcal{H}|$ cubes centered in p with edge lengths in $\mathcal{H}$ (using Algorithm 4). Then, we perform TCAM lookups with cubes of increasing edge length values until a match is found. As in the previous method, if a point was matched with a query of edge length h, then it is a solution of the $h/2$-Near Neighbor Report Problem.

4.1.1 Analysis of Approximation

Let $\mathcal{H} = \{h_1, h_2, \ldots, h_{max}\}$ such that $h_i < h_j$ for each $i < j$. Matching a data point p corresponding to a cube with edge length h_i implies that $D(p, q) \leq \left\lfloor \frac{h_i}{2} \right\rfloor$ (where D is defined under ℓ_∞). Since h_i is the first edge length to be matched, $D(s^*, q) \geq \left\lfloor \frac{h_{i-1}}{2} \right\rfloor + 1$ (s^* is the exact nearest neighbor of q). This implies that under ℓ_∞, both methods solve the c-Approximate-Nearest-Neighbor Problem for $c = \max_{h_i \in \mathcal{H}} \frac{\lfloor h_i/2 \rfloor}{\lfloor h_{i-1}/2 \rfloor + 1}$, where h_i is the i^{th} smallest element in $\mathcal{H}$ and $h_1 = 1 \in \mathcal{H}$. [4]

Algorithm 5 Exact k-Nearest Neighbors Search Algorithm in ℓ_p

```
1: function FIND-EXACT-KNN(q, S, k)
2:     N = ∅                                  ▷ Candidate neighbors set
3:     h_last = −1
4:     repeat
5:         (s, h) ← TCAMLOOKUP(q, S)
           ▷ returns the datapoint and correponding edge length
6:         if |N| < k or h_last = −1 or h = h_last then
7:             N ← N ∪ 𝒩(s, h)
8:             h_last ← h
9:         end if
10:    until |N| ⩾ k and h > h_last
11:    R ← arg min^k_{s'∈N} D_p(s', q)        ▷ k min-distance points
12:    return R
13: end function
```

In order to get the exact nearest neighbor in ℓ_∞, one can choose $\mathcal{H}$ to be the set of odd numbers. Reducing the size of $\mathcal{H}$ reduces the number of required entries, but decreases the quality of the results. For example, to get a c-approximate solution, $\mathcal{H}$ can consist of all even values up to $2/(c-1)$, along with the values of a geometric series starting at $2/(c-1)$, with the parameter c.

When distances are defined under ℓ_p norms, for finite values of $p \geq 1$, the approximation ratio is at most $c \cdot \sqrt[p]{d}$ in ℓ_p, where c is the approximation ratio in ℓ_∞.

4.1.2 Database Update

Our algorithms allow efficient hot updates in the lookup database (the set S). Deletion of data points is trivial (simply delete all corresponding entries from the TCAM). When using the *time-efficient method*, efficient addition of new points is possible by keeping some empty TCAM entries between entries of different edge length, by adding entries for the corresponding cubes in these empty slots. Also, one can track deletion for more empty slots. Nevertheless, this further increases space requirement.

When using the *memory-efficient method*, the situation is simpler: since the order of entries is not important, point addition or deletion requires a single TCAM entry update.

4.2 Exact Nearest-Neighbor Search in ℓ_p

Our algorithms achieve $\sqrt[p]{d}$ approximate solution under ℓ_p norm. We suggest the following extension to find (exactly) the nearest neighbor under ℓ_p: For each data point $s \in S$ and for each edge length $h \in \mathcal{H}$, we *precompute* the *neighborhood set* $\mathcal{N}(s, h) = \{s' \in S \mid D_p(s, s') \leq h \sqrt[p]{d}\}$, where D_p is the distance between the two points under the p-norm. The neighborhood sets are stored in memory. Precomputing these sets is possible since datasets are relatively static and the neighborhood sets do not depend on the query points.

Since for every two points, the distance in ℓ_p is at most $\sqrt[p]{d}$ the distance in ℓ_∞, we immediately conclude that if the algorithms described in Section 4.1 return a data point s for query point q with some distance $h \in \mathcal{H}$, then the exact nearest neighbor in ℓ_p of q is in $\mathcal{N}(s, h)$.

While this method requires additional computations following the TCAM lookup, in most datasets the number of

[4] To get a bounded approximation ratio, 1 must be added to $\mathcal{H}$.

points in $\mathcal{N}(s,h)$ would be very small. In our experiments (see Section 5) $\mathcal{N}(s,h)$ contained only s itself for lower values of h in most cases and was small even for higher values of h. Thus, the time required to find the exact nearest neighbor is still much shorter than that required for brute-force over all points in the database.

4.2.1 k–Nearest-Neighbors Search

The precomputed neighborhood sets can also be used to find k-nearest neighbors instead of only one. However, the number of neighbors in these sets might be smaller than k, so one TCAM lookup might not suffice. Algorithm 5 describes the required process, assuming usage of multi-match techniques such as the one presented in [14].

4.3 Algorithms for the Partial Match Problem

The PARTIAL MATCH PROBLEM is defined as follows:

DEFINITION 6. *Given a set of data points $S = \{p_i\}_{i=1}^n$, $p_i \in \mathbb{Z}^d$, a query point $q \in \mathbb{Z}^d$, and a subset of the dimensions $D_q \subseteq \{1,\ldots,d\}$ of size $d_q < d$,* THE PARTIAL MATCH PROBLEM *is to find the point $s^* = \arg\min_{s' \in S} D(s',q)|_{D_q}$, where $D(a,b)|_{D_q}$ is the distance between a and b under some norm in the d_q-dimensional space. Namely, for a p-norm,*

$$D(a,b)|_{D_q} = \left(\sum_{i \in D_q} |a_i - b_i|^p \right)^{1/p}.$$

In traditional computing models the PARTIAL MATCH PROBLEM is known to be *more difficult* [6] than the nearest neighbor problem, where all relevant dimensions are given apriori. For example, LSH (and its extension to TCAMs, TLSH [26]) cannot be used to solve this problem. However, our solution for the NN problem can be used instantly to solve the partial match problem.

Under the maximum norm ℓ_∞, a PARTIAL MATCH solution is to replace, *in the queries*, all the bits corresponding to coordinates in irrelevant dimensions with * bits. We replace coordinates in queries and not for data point, as the relevant dimensions are selected per query. This technique works both for our *time-efficient* and *memory-efficient methods*.

For ℓ_p, our solution results in $\sqrt[p]{d_q}$ approximation, where d_q is the dimension of the specific query. The extensions to EXACT NEAREST NEIGHBOR SEARCH and k–NEAREST NEIGHBORS SEARCH, as described in Section 4.2, work also for this problem. The neighborhood sets are precomputed on the d-dimensional space, but queries and distance computations after queries are done on the specific d_q dimensional space. The results are still correct as distances in the d_q-dimensional space are bounded by distances in the d-dimensional space.

4.4 k-Means Clustering on TCAM

Another closely related problem that could benefit from using TCAM with our technique is high-dimensional geometric clustering. The k-MEANS CLUSTERING problem, for example, is usually solved as a sequence of nearest-neighbor search problems, each of these consists of a database with k d-dimensional points [16].

Algorithm 6 shows how the traditional k-MEANS CLUSTERING algorithm can be implemented on TCAM using our encoding function `tcode` , under ℓ_∞ norm. As k is usually much smaller than the number of points, this solution requires relatively low TCAM space. Still, a standard TCAM can support up to thousands of clusters using this algorithm.

Algorithm 6 k-MEANS CLUSTERING Algorithm on TCAM

```
1: function FIND-K-MEANS(S, k)
2:     changed ← false
3:     t ← 1
4:     Randomly select D ← {d_1, ..., d_k} ⊆ S (|D| = k)
5:     for i ← 1 to k do
6:         C_0^i ← ∅
7:         C_1^i ← ∅
8:     end for
9:     repeat
10:        Clear TCAM
11:        for h ← 1 to H do
12:            for i ← 1 to k do
13:                Add tcode (d_i, h) to end of TCAM
14:            end for
15:        end for
16:        for each s ∈ S do
17:            d_i ← query TCAM with tcode (s)
18:            if s ∉ C_{t-1}^i then
19:                changed ← true
20:                C_t^i ← C_t^i ∪ {s}
21:            end if
22:        end for
23:        if changed then
24:            for i ← 1 to k do
25:                d_i ← center of C_t^i
26:                C_{t+1}^i ← ∅
27:            end for
28:            t ← t + 1
29:            changed ← false
30:        end if
31:    until changed = false
32:    return D = {d_1, ..., d_k}
33: end function
```

5. EVALUATION AND EXPERIMENTAL RESULTS

We evaluate our nearest-neighbor algorithms using an image similarity search application (using GIST [19] descriptors), on a real-life image dataset [24]. We then compare the results and performance with prior-art solutions. Our evaluation is based both on experiments with real-life TCAM devices and simulations. Each image in the dataset was encoded as a GIST vector in $\mathbb{R}^{512}$, downsampled to $\mathbb{R}^{40}$ and quantified to $\{0,\ldots,255\}^{40}$ before performing search. Images were randomly partitioned to a dataset of $21,019$ images a query set of $1,000$ images.

5.1 Experiment with a TCAM Device

Since there is no evaluation board for such devices, we used a commodity network switch (Quanta T1048-LB9) that contains a TCAM for our experiment (similarly to [26]). This switch has 48 1 Gbps ports, each handling at most $1.5M$ packets per second. TCAM is used for packet classification for OpenFlow 1.3 [20]. Using the OpenFlow interface to the switch, we mapped each entry produced by our algorithms to a set of header fields. A commercial traffic generator injected manually crafted packets that contain the queries in their headers, where each query is broken into header fields in the same way TCAM entries are stored.

We verified correctness by counting the number of matches of each TCAM entry. Using one ingress port the switch easily achieved a throughput of $1.5M$ queries per second,

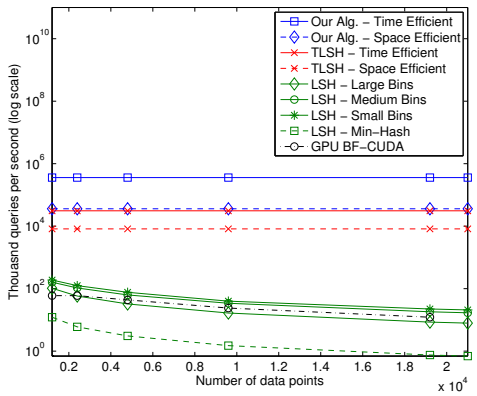

Figure 7: Throughput comparison of the various algorithms for solving the nearest neighbor problem, based on $360MHz$ TCAM throughput. We denote TLSH with one TCAM lookup per r-Near Neighbor Report Problem instance as *time-efficient*, and TLSH with $\log(1/\varepsilon)$ TCAM lookups per instance as *space-efficient*.

which is the upper bound of the link between the traffic generator and the switch (but not of the TCAM). Using 24 ingress ports we achieved throughput of $35.69M$ queries per second (almost $1.5M \times 24$). Hence, the bottleneck was not in the TCAM: If we had more ports we could have reached much higher throughput.

5.2 Simulation Results

We compared our results to the results of brute-force exact nearest neighbor (using MATLAB or on GPU [9]), locality sensitive hashing [2,12] (using implementation from [1]), and TLSH [26]. LSH approximated results in ℓ_2 were comparable to our approximated results in ℓ_2 only when LSH used the most complex hash functions, or when used very large bins. Both options mean longer computation time due to either more complex hash computation or much more distance computations. A comparison of the throughput achieved by these algorithms is shown in Figure 7. [5]

The required TCAM space for our *space efficient* method is $8M$ bits and for our *time efficient* method with 10 different cube sizes is $80M$ bits, with 440 bits wide entries. These requirements are available in most modern TCAM devices. As discussed in Section 5.2.1, TLSH requires much higher TCAM capacity and much wider TCAM entries.

5.2.1 Comparison to TLSH

As recalled, Shinde et al. [26] were the first to suggest using TCAM for nearest neighbor search. They use a ternary variant of the Locality Sensitive Hashing, called TLSH, to provide a probabilistic solution for the nearest neighbor problem. The main advantages of our algorithms over TLSH are that our time-efficient algorithm solves *multiple* instances of the r-Near-Neighbor Report Problem in a single TCAM lookup, while TLSH requires $|\mathcal{H}|$ lookups (hence the factor of 10 difference in the results presented in Figure 7), and that the TCAM space requirements, and specifically and more importantly TCAM entry width requirement, are lower in at least one order of magnitude than those of TLSH. Furthermore, our algorithms provide deterministic results and are not subject to probabilistic errors, and they allow database hot updates.

[5]LSH implementations were ran on an Intel Core i7 2600 3.4GHz CPU. We used the same dataset and queries for our algorithms, LSH and TLSH. TCAM algorithms used 10 different cube sizes. GPU throughput is as reported in [9] for the closest lower values of n and d.

6. CONCLUSIONS

TCAM devices introduce high parallelism and superior expressiveness. In networking, TCAMs can be found in most moderate and large size switches and routers, due to their ability to classify packets in a constant lookup time and linear memory size. This ability makes it possible to outperform known lower bounds in traditional memory models.

This work shows that TCAM can also provide such breakthrough results for the nearest neighbor search problem, which is a crucial problem in many fields of Computer Science and specifically in contemporary database systems. Including TCAMs in real-time devices can provide a nearest neighbor search rate of more than one billion queries per second. TCAM chips become more popular and are constantly improving, so our solution is expected to further improve in the near future.

Acknowledgments

This research was supported by the European Research Council under the European Union's Seventh Framework Programme (FP7/2007-2013)/ERC Grant agreement n° 259085.

7. REFERENCES

[1] M. Aly, M. Munich, and P. Perona. Indexing in large scale image collections: Scaling properties and benchmark. In *WACV*, pages 418–425, 2011.

[2] A. Andoni and P. Indyk. Near-optimal hashing algorithms for approximate nearest neighbor in high dimensions. *Comm. of the ACM*, 51(1):117–122, 2008.

[3] N. Beckmann, H.-P. Kriegel, R. Schneider, and B. Seeger. The r*-tree: An efficient and robust access method for points and rectangles. In *SIGMOD*, pages 322–331, 1990.

[4] J. S. Beis and D. G. Lowe. Shape indexing using approximate nearest-neighbour search in high-dimensional spaces. In *CVPR*, pages 1000–1006, 1997.

[5] J. L. Bentley. Multidimensional divide-and-conquer. *Comm. of the ACM*, 23(4):214–229, 1980.

[6] A. Borodin, R. Ostrovsky, and Y. Rabani. Lower bounds for high dimensional nearest neighbor search and related problems. In *STOC*, pages 312–321, 1999.

[7] M. Brown and D. G. Lowe. Recognising panoramas. In *ICCV*, volume 3, page 1218, 2003.

[8] I. Corp. Intel Xeon processor E7-4870, 2011. `http://ark.intel.com/products/53579/`.

[9] V. Garcia, E. Debreuve, F. Nielsen, and M. Barlaud. K-nearest neighbor search: Fast GPU-based implementations and application to high-dimensional feature matching. In *ICIP*, pages 3757–3760, 2010.

[10] F. Gray. Pulse code communication. US Patent 2,632,058, March 17 1953 (filed November 13 1947).

[11] C. Inc. NEURON search processors, 2014. `http://bit.ly/1uSaH8q`.

[12] P. Indyk and R. Motwani. Approximate nearest neighbors: Towards removing the curse of dimensionality. In *STOC*, pages 604–613, 1998.

[13] W. Jiang, Q. Wang, and V. K. Prasanna. Beyond TCAMs: An SRAM-based parallel multi-pipeline architecture for terabit IP lookup. In *INFOCOM*, pages 1786–1794, 2008.

[14] K. Lakshminarayanan, A. Rangarajan, and S. Venkatachary. Algorithms for advanced packet classification with ternary CAMs. In *SIGCOMM*, pages 193–204, 2005.

[15] L. Liang, C. Liu, Y.-Q. Xu, B. Guo, and H.-Y. Shum. Real-time texture synthesis by patch-based sampling. *ACM ToG*, 20(3):127–150, 2001.

[16] S. Lloyd. Least squares quantization in pcm. *IEEE Trans. Inf. Theor.*, 28(2):129–137, Sep 2006.

[17] D. G. Lowe. Object recognition from local scale-invariant features. In *ICCV*, pages 1150–1157, 1999.

[18] Nvidia. Tesla K80 GPU accelerator, Nov. 2014. `http://bit.ly/1C69bVd`.

[19] A. Oliva and A. Torralba. Modeling the shape of the scene: A holistic representation of the spatial envelope. *IJCV*, 42:145–175, 2001.

[20] Open Networking Foundation. *OpenFlow Switch Specification Version 1.3.2*, April 2013.

[21] J. Philbin, O. Chum, M. Isard, J. Sivic, and A. Zisserman. Object retrieval with large vocabularies and fast spatial matching. In *CVPR*, pages 1–8, 2007.

[22] V. Ravikumar and R. N. Mahapatra. TCAM architecture for IP lookup using prefix properties. *Micro, IEEE*, 24(2):60–69, 2004.

[23] Renesas Electronics America Inc. 20Mbit QUAD-search content addressable memory. `http://bit.ly/18hYySx`.

[24] B. C. Russell, A. Torralba, K. P. Murphy, and W. T. Freeman. Labelme: A database and web-based tool for image annotation. *IJCV*, 77(1-3):157–173, 2008.

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

[26] R. Shinde, A. Goel, P. Gupta, and D. Dutta. Similarity search and locality sensitive hashing using ternary content addressable memories. In *SIGMOD*, pages 375–386, 2010.

[27] R. Weber, H.-J. Schek, and S. Blott. A quantitative analysis and performance study for similarity-search methods in high-dimensional spaces. In *VLDB*, pages 194–205, 1998.

APPENDIX

A. PROOFS

THEOREM 1. *If all points are BRGC-encoded, then a single ternary BRGC codeword suffices to admissibly encode an interval $I = [x, y \bmod 2^w)$ if and only if there exist non-negative integers i, k, for which $x = i \cdot 2^k$ and $y = (i+2) \cdot 2^k$. Specifically, one of the following cases holds:*

1. *If i is even, the $(k+1)$ least significant bits of the codeword are $*$, and the rest are either 0 or 1. Thus, the ternary codeword is $(k+1)$-prefix.*
2. *If i is odd, the k least significant bits of the codeword are $*$, one additional bit is $*$, and the rest are either 0 or 1. Thus, the ternary codeword is k-semi-prefix.*

PROOF. The proof follows by induction on k: For $k = 0$, intervals are $[i, i+2)$. These intervals are simply two adjacent leaves in the BRGC tree representation, and, by definition of Gray code, they differ in a single bit only.

If i is even, then there is an even number of leaves before these two, and thus these two leaves are siblings and have a common direct ancestor x with height $k+1 = 1$. Thus, the $(k+1)$-prefix that is the concatenation of the $\log(w) - 1$ bits that represent the path to x with one $*$, represents the interval $[i, i+2)$ (case 1 in Theorem 1).

If i is odd, then the two leaves do not have a common direct ancestor. However, they do have some common ancestor up in the higher levels of the tree, let it be at height h. Thus, their representation differ in the h^{th} bit. Since the codewords of i and $i+1$ differ only in one bit, there are no additional bits where they differ. Therefore, the k-semi-prefix in which the h^{th} bit is $*$ and the rest of the bits are as in i and $i+1$ BRGC codewords, represents the interval $[i, i+2)$ (case 2 in Theorem 1).

We assume that the lemma is correct for some k, and show that it is correct also for $k+1$: Let $I = [i \cdot 2^{k+1}, (i+2) \cdot 2^{k+1})$ be an interval of length 2^{k+2}, for some positive integer i. Let $j_1 = 2i, j_2 = 2i+2$, and let $I_1 = [j_1 \cdot 2^k, (j_1+2) \cdot 2^k), I_2 = [j_2 \cdot 2^k, (j_2+2) \cdot 2^k)$ be two intervals of length 2^{k+1}. Then, $I = I_1 \cup I_2$, and since j_1 and j_2 are even, I_1, I_2 can be represented using $(k+1)$-prefixes (case 1 in Theorem 1).

If i is even, then in the tree representation of the BRGC encoding there exists an even number of subtrees of height $k+1$ before the subtree that represents I_1. Thus, the roots of the subtrees that represent I_1, I_2 are siblings, and the first $w-k-2$ bits of their ternary BRGC codewords are equal. Wildcarding the $k+2$ least significant bit would yield a $(k+2)$-prefix that represents their union and thus represents I.

If i is odd, then the roots of the subtrees that represent I_1, I_2 are not siblings, but they do have some common ancestor. We denote the right bound of I_1 as $x = (2i+2) \cdot 2^k - 1$ and the left bound of I_2 as $y = (2i+2) \cdot 2^k$. x and y are two consecutive integers and thus their BRGC representation differ in one bit only. If we assume towards a contradiction that the difference is in one of the $k+1$ least significant bits, then both BRGC codewords of these points share the same prefix of length that is greater than $w-k-1$, so the two points can be represented using the same subtree of height $k+1$, which is impossible as $I_1 \neq I_2$. Thus, the difference between the two codewords is in one of the $w-k-1$ most significant bits, and the $(k+1)$-semi-prefix that has '$*$'s in this bit and in the $k+1$ least significant bits, represents the

union of I_1 and I_2 which is I. □

LEMMA 1. *For any interval $I = [x, x + h)$ of length $h = 2^k$ ($k \in \mathbb{N}_0$), $\texttt{cover}(I)$ fully contains I.*

PROOF. Assume by contradiction that I starts before $\texttt{cover}(I)$ starts or ends after $\texttt{cover}(I)$ ends. If I starts before $\texttt{cover}(I)$, $x < \lfloor x/h \rfloor \cdot h$. So $x/h < \lfloor x/h \rfloor$, which is of course impossible. Also, if I ends before $\texttt{cover}(I)$ ends, $x + h > (\lfloor x/h \rfloor + 2) \cdot h$. This implies that $x/h > \lfloor x/h \rfloor + 1$ which is also impossible, and thus a contradiction. □

LEMMA 2. *If an interval I is nontrivial, then no other interval from the same layer is fully contained in $\texttt{cover}(I)$.*

PROOF. Assume $I = [x, x + h)$, where $h = 2^k$, and that there exists another interval from the same layer, I', that is also fully contained in $\texttt{cover}(I)$. By the definition of the layers, I and I' are both of length h and are not overlapping. By Definition 5, $\texttt{cover}(I)$ is of length 2^{k+1}, implying that the union of I and I' is equal to $\texttt{cover}(I)$, but since both intervals are fully contained in the cover, the union is exactly the cover. This, in turn, implies that $x \bmod h = \lfloor x/h \rfloor$. Choosing $i = 2 \lfloor x/h \rfloor$ yields that $I = [i \cdot 2^k, (i + 2) \cdot 2^k)$ and thus, by Theorem 1, I is a trivial interval, in contrast to the assumption. □

LEMMA 3. *Let $I = [x, x+h)$, where $h = 2^k$, be an interval in L_h^i. For every point p in $\texttt{cover}(I)$, if $p \in I$, p has the same bit value as I, and if $p \notin I$, then p has the opposite bit value.*

PROOF. Assume that a point $p \in I$, has a bit value that is different than the bit value of I. p is in I so $p - x \leqslant h$, and thus $\lfloor \frac{p-i}{h} \rfloor = \lfloor \frac{p-i}{h} \rfloor$, meaning that the bit value of p must be equal to the bit value of I.

To prove the other direction, assume that a point $p \notin I$ has the same bit value as I. Let $I_{\text{before}} = [x - h, x)$ be the interval that precedes I in L_h^i and $I_{\text{after}} = [x+h, x+2h)$ be the interval that succeeds I in L_h^i. The bit value of I_{before} and I_{after} must be different than the bit value of I as they are both adjacent to I. Since $p \in \texttt{cover}(I)$ but not in I, and since the length of $\texttt{cover}(I)$ is at most $2h$, p must be either in I_{before} or in I_{after}, and thus it must have the opposite bit value than I. □

THEOREM 2. *The function $\texttt{tcode}$ is an admissible encoding function for intervals of length $h = 2^k$.*

PROOF. Assume that there exist a point p and an interval I for which $p \in I$ but $\texttt{tcode}(p) \not\approx \texttt{tcode}(I)$. p is in I so the ternary BRGC of I (in case I is trivial) or of $\texttt{cover}(I)$ (in case I is nontrivial) must match the BRGC encoding of p ternary-wise. Thus, some extra bit does not match. Since for trivial intervals all extra bits are $\texttt{*}$, I must be nontrivial. For nontrivial intervals, only one extra bit in $\texttt{tcode}(I)$ is not a '$\texttt{*}$'. However, this bit must be equal to the corresponding bit in $\texttt{tcode}(p)$ by Lemma 3, which is a contradiction to the assumption that $\texttt{tcode}(p) \not\approx \texttt{tcode}(I)$.

To prove the opposite direction, assume that there exist a point p and an interval I for which $\texttt{tcode}(p) \approx \texttt{tcode}(I)$ but $p \notin I$. The BRGC encoding of p must match the ternary BRGC encoding of I (in case I is trivial) or $\texttt{cover}(I)$ (if I is nontrivial). If I is trivial and there is a match then $p \in I$, as all extra bits in $\texttt{tcode}(I)$ are '$\texttt{*}$'. Thus, I must be nontrivial, and p must be inside $\texttt{cover}(I)$. However by Lemma 3, if $p \in \texttt{cover}(I)$ and has the same bit value as I for the layer I belongs to, then p must be in I. □

LEMMA 4. *For any point $p \in \mathcal{U}$ and any two intervals $I_1, I_2 \subseteq \mathcal{U}$, if $\texttt{tcode}$ is an admissible encoding function for I_1 and I_2, then $\texttt{tcode}(p) \approx \texttt{tcode}(I_1) \sqcap \texttt{tcode}(I_2)$ if and only if $p \in I_1 \cap I_2$.*

PROOF. Assume $\texttt{tcode}(p) \approx \texttt{tcode}(L_1) \sqcap \texttt{tcode}(L_2)$ and $p \notin I_1 \cap I_2$. Without loss of generality, assume $p \notin I_1$. Since $p \notin I_1$ and $\texttt{tcode}$ is an admissible encoding function, there exists some i for which $\texttt{tcode}(p)_i \neq \texttt{*}$, $\texttt{tcode}(I_1)_i \neq \texttt{*}$, and $\texttt{tcode}(p)_i \neq \texttt{tcode}(I_1)_i$. Without loss of generality, let $\texttt{tcode}(p)_i = 0$, so $\texttt{tcode}(I_1)_i = 1$ and therefore $\texttt{tcode}(I_1)_i \sqcap \texttt{tcode}(I_2)_i$ is either 1 or $\perp$. Thus, by definition, $\texttt{tcode}(p)_i \not\approx \texttt{tcode}(I_1)_i \sqcap \texttt{tcode}(I_2)_i$ implying $\texttt{tcode}(p) \not\approx \texttt{tcode}(I_1) \sqcap \texttt{tcode}(I_2)$, which is a contradiction.

To prove the other direction, assume that $p \in I_1 \cap I_2$. Since $p \in I_1$ and $p \in I_2$ and $\texttt{tcode}$ is admissible, $\texttt{tcode}(p)_i \approx \texttt{tcode}(I_1)_i$ and $\texttt{tcode}(p)_i \approx \texttt{tcode}(I_2)_i$, for any i. The admissibility of $\texttt{tcode}$ also implies that $\texttt{tcode}(p)_i$ is either 0 or 1. Assume without loss of generality that for some i it is 0. Then, $\texttt{tcode}(I_1)_i$ and $\texttt{tcode}(I_2)_i$ are either 0 or $\texttt{*}$. Hence, $\texttt{tcode}(I_1)_i \sqcap \texttt{tcode}(I_2)_i$ is either 0 or $\texttt{*}$, and therefore, $\texttt{tcode}(p)_i \approx \texttt{tcode}(I_1)_i \sqcap \texttt{tcode}(I_2)_i$. Since this is true for any i, it implies that $\texttt{tcode}(p) \approx \texttt{tcode}(I_1) \sqcap \texttt{tcode}(I_2)$, and the claim follows. □

LEMMA 5. *If $\texttt{tcode}$ is an admissible encoding function for I_1 and I_2, and the result of $\texttt{tcode}(I_1) \sqcap \texttt{tcode}(I_2)$ is $\perp$ then $I_1 \cap I_2 = \emptyset$.*

PROOF. Assume $\texttt{tcode}(I_1) \sqcap \texttt{tcode}(I_2) = \perp$ and to the contrary, that $I_1 \cap I_2 \neq \emptyset$. Then, there exists some i for which, without loss of generality, $\texttt{tcode}(I_1)_i = 0$ and $\texttt{tcode}(I_2)_i = 1$, and some point $p \in I_1 \cap I_2$. From the admissibility of $\texttt{tcode}$, if $p \in I_1$, then $\texttt{tcode}(p) = 0$, and thus $p \notin I_2$, and if $p \in I_2$, then $\texttt{tcode}(p) = 1$, and thus $p \notin I_1$, which is a contradiction. □

Note that the other direction of Lemma 5 is not necessarily true: the conjunction of codes of two disjoint intervals may not be $\perp$.