Improved Explicit Data Structures in the Bit-Probe Model Using Error-Correcting Codes

Palash Dey
Department of Computer Science and Engineering, Indian Institute of Technology Kharagpur, India
https://cse.iitkgp.ac.in/~palash/
palash.dey@cse.iitkgp.ac.in

Jaikumar Radhakrishnan
School of Technology and Computer Science, Tata Institute of Fundamental Research, Mumbai, India
https://www.tcs.tifr.res.in/~jaikumar
jaikumar@tifr.res.in

Santhoshini Velusamy
School of Engineering and Applied Sciences, Harvard University, Cambridge, MA, USA
https://scholar.harvard.edu/santhoshiniv/home
svelusamy@g.harvard.edu

Abstract

We consider the bit-probe complexity of the set membership problem: represent an n-element subset S of an m-element universe as a succinct bit vector so that membership queries of the form "Is $x \in S$" can be answered using at most t probes into the bit vector. Let $s(m, n, t)$ (resp. $s_N(m, n, t)$) denote the minimum number of bits of storage needed when the probes are adaptive (resp. non-adaptive). Lewenstein, Munro, Nicholson, and Raman (ESA 2014) obtain fully-explicit schemes that show that

$$s(m, n, t) = \mathcal{O}((2^t - 1)m^{1/(t-\min\{2\lfloor \log n \rfloor, n-3/2\})})$$

for $n \geq 2, t \geq \lfloor \log n \rfloor + 1$.

In this work, we improve this bound when the probes are allowed to be superlinear in n, i.e., when $t \geq \Omega(n \log n)$, $n \geq 2$, we design fully-explicit schemes that show that

$$s(m, n, t) = \mathcal{O}((2^t - 1)m^{1/(t-\frac{n-1}{2^{t/(2(n-1))}})}),$$

asymptotically (in the exponent of m) close to the non-explicit upper bound on $s(m, n, t)$ derived by Radhakrishan, Shah, and Shannigrahi (ESA 2010), for constant n.

In the non-adaptive setting, it was shown by Garg and Radhakrishnan (STACS 2017) that for a large constant n_0, for $n \geq n_0$, $s_N(m, n, 3) \geq \sqrt{mn}$. We improve this result by showing that the same lower bound holds even for storing sets of size 2, i.e., $s_N(m, 2, 3) \geq \Omega(\sqrt{m})$.

2012 ACM Subject Classification Theory of computation → Data compression; Theory of computation → Cell probe models and lower bounds

Keywords and phrases Set membership, Bit-probe model, Fully-explicit data structures, Adaptive data structures, Error-correcting codes

Digital Object Identifier 10.4230/LIPIcs.MFCS.2020.28

Funding *Palash Dey*: This work was done when the author was a post-doctoral Visiting Fellow at TIFR, Mumbai. This work was supported by the Department of Science and Technology, Government of India, under the INSPIRE Faculty Scheme no. DST/INSPIRE/04/2016/001479.
Jaikumar Radhakrishnan: This work was supported by the Department of Atomic Energy, Government of India, under project no. 12-R&D-TFR-5.01-0500; part of this work was done while the author was visiting the Chinese University of Hong Kong.
Santhoshini Velusamy: Most of this work was done while the author was at TIFR, Mumbai, for the Visiting Students' Research Programme (VSRP) in 2017; part of this work was supported by a Simons Investigator Award and the NSF Award CCF 1715187.

Acknowledgements The authors would like to thank Madhu Sudan and the anonymous referees for several useful suggestions that helped improve the presentation of this paper.

© Palash Dey, Jaikumar Radhakrishnan, and Santhoshini Velusamy;
licensed under Creative Commons License CC-BY

1 Introduction

In this paper, we consider the classical static set membership problem: given an n-element subset S of the universe $[m]$, represent it succinctly in memory as a sequence of bits, so that membership queries of the form "Is x in S?" can be answered efficiently. How succinct can such a data structure be? The number of subsets is $\binom{m}{n}$ and clearly, each set must have a different representation, so the data structure must use at least $\log \binom{m}{n} = \Theta(n \log(m/n))$ bits (we assume $n \ll m$). Note that the standard data structure which stores the set as a sorted table and uses binary search to answer queries uses not much more space than this lower bound (ignoring constant factors). This data structure reads about $(\log n)(\log m)$ bits to answer membership queries. Can the membership queries be more efficient?

Several works have addressed this question. E.g., Fredman, Komlós and Szemerédi [6], Brodnik, and Munro [4], Pagh [17] and Patrascu [18] construct data structures that use close to $\log \binom{m}{n}$ bits of memory organized in $(\log m)$-bit words and yet need to read only a constant number of words (i.e., $O(\log m)$ bits in all) to answer membership queries. It can also be shown that $\Omega(\log m)$ bits must be read to answer queries if the data structure is restricted to use $O(n \log(m/n))$ space. Thus, the problem is well settled in the realistic cell probe model.

The trade-off between the space needed and the *number of probes* is not completely settled in the *bit probe model*. The problem originated in the work of Minsky and Papert [15], who considered representations of sets as a sequence of bits so that membership can be determined by reading a small number of bits *on average*. The recent interest in the problem can be traced to Buhrman, Miltersen, Radhakrishnan, and Venkatesh [5], who showed that there exist randomized schemes that answer membership queries with *just one* bit probe (while erring with some small probability). The survey article of Nicholson, Raman, and Rao [16] describes several other data structure problems that have been addressed in the bit probe model.

The set membership problem can also be viewed as a problem of data compression. An n-element subset of $[m]$ can be viewed as a sparse binary string of length m. The trade-off that we have in the set membership problem then corresponds to the relationship between compression that can be achieved and the number of bits of the compressed string that need to be read to recover a bit of the original sparse string. For recent work in the information-theoretic literature related to this, see Makhdoumi, Huang, Médard, Polyanskiy [14]. In practice, Bloom filters [2] offer a solution for the set membership problem: with a small number of bit probes they determine if an element is in the set; however, they do not guarantee correct answers always, for they allow a small number of false positives. In this paper, we are looking for solutions that allow no errors.

To place the contributions of this paper in the context of previous work on the set membership problem, we let $s(m, n, t)$ denote the minimum number of bits needed to represent set S of size at most n from the universe $[m]$ so that membership queries of the form "Is x in S?" can be answered using at most t bit probes. We will be interested in the setting where $n \ll m$. Buhrman *et al.* [5] showed that $s(m, n, t) = \Omega(tn^{1-1/t}m^{1/t})$. The current best general upper bound, due to Garg and Radhakrishnan [9], gives $s(m, n, t) = O(\exp(e^{2t})n^{1-2/(t+1)}m^{2/(t+1)} \log m)$ (assuming $n = m^{1-\epsilon}$ for $\epsilon > 0$ and $t \geq 3$ and $t \leq (\log \log m)/10$). We restrict attention to situations where n and t are small (large constants), in which case the most significant difference between the lower and upper bounds is the exponent of m: we have $1/t$ in the lower bound and $2/(t + 1)$ in the upper bound.

For $n = 1$, it is straightforward to show that $s(m, 1, t) = O(m^{1/t})$. We do not, however, have tight bounds for larger t. The best bounds we know for the case $n = 2$ and $t = 2$ are

$$c_1 m^{4/7} \leq s(m, 2, 2) \leq c_2 m^{2/3},$$

where c_1 and c_2 are constants [23, 22]. Remarkably, it is known that $s(m, 3, 2) = \theta(m^{2/3})$: the upper bound was shown by Baig and Kesh [1] and the lower bound was shown by Kesh [12].

The works of Radhakrishnan, Shannigrahi, and Shah [23] and Lewenstein, Munro, Nicholson, and Raman [13] address the question by constructing schemes showing that for a certain range of parameters ($t \gg n$), the exponent of m in $s(m, n, t)$ comes close to $1/t$. Radhakrishnan *et al.* used a probabilistic argument to show that

$$s(m, n, t) = O(ntm^{1/(t-nt2^{-t+1})}), \tag{1}$$

for $t > n \geq 2$.

Later, Lewenstein *et al.* built *fully-explicit* schemes that show that

$$s(m, n, t) = O((2^t - 1)m^{1/(t-\min\{2\lfloor \log n \rfloor, n-3/2\})}), \tag{2}$$

for $n \geq 2$ and $t \geq 2\lceil \log n \rceil + 1$. A scheme is fully-explicit if the locations of the probes to be made are computable in time polynomial in t and $\log m$ given the query, and the bits to be stored in the data structure are computable in time polynomial of its size, given the subset to be stored [5]. Though the bound in (2) is weaker than the bound in (1) for $t \gg n$, it is based on fully-explicit constructions. They obtained an explicit scheme showing that $s(m, 2, 3) = O(m^{2/5})$ (the same bound was shown by Radhakrishnan *et al.*, but the proof was based on non-explicit existence arguments [23]). Note that in this bound the exponent of m is of the form $1/t + \Theta(\log n/t^2)$ when $t \gg \log n$. We construct a fully-explicit scheme with better space complexity than Equation (2).

▶ **Theorem 1** (Result 1). $s(m, n, t) = \mathcal{O}((2^t - 1)m^{1/(t-\frac{n-1}{2^{t/(2(n-1))}})})$

The exponent of m in our bound has the form $1/t + (n-1) \cdot 2^{-t/(2(n-1))}/t^2$. (For example, if we set $t = n^2 \log\log m$, we get that $s(m, n, t) = O(m^{1/t})$, whereas the RHS of (2) is $\omega(m^{1/t})$.) For $n = 2$, however, Lewenstein *et al.* obtain a bound of the form

$$s(m, n, t) = O((2^t - 1)m^{1/(t-2^{2-t})} \tag{3}$$

However, they do not present a comparable generalization for larger n. Our Theorem 1 can be seen as an analog of (3) for $n > 2$. We now describe at a high level the relationship between our work and that of Lewenstein *et al.* [13]. If only about t bit probes are allowed, it is natural to split the universe into blocks of size approximately $m^{1/t}$ and build a tree over them, with each internal node having degree approximately $m^{1/t}$. At each internal node, an array of size $m^{1/t}$ helps determine which path down the tree a particular query must take. While deriving (3), Lewenstein *et al.* observed that there is some choice available in the way the various arrays are populated. This redundancy can be used to provide some more information to the query algorithm. They further used the fact that if exactly one of several bits is 1, then their parity is also 1, that is, one parity bit is enough to detect one error. All this points to the possibility that error-correcting codes may have a role to play in the design of such data structures. Our construction makes direct use of error correcting codes to exploit the freedom available.

Apart from the above general results, we focus attention on query schemes that make three *non-adaptive* probes, i.e., the probes are made in parallel. It is not known if such schemes can give significant savings over the characteristic vector for large sets (if $n \gg \log n$). Let $s_N(m, n, t)$ denote the minimum number of bits needed to represent sets S of size at most n from the universe $[m]$ so that membership queries of the form "Is x in S?" can be answered using at most t non-adaptive bit probes. The most efficient schemes we know for small n are obtained using the inequality $s_N(m, n, 3) \leq s(m, n, 2)$ and appealing to the fact that any two-probe adaptive scheme can be implemented as a three-probe non-adaptive scheme. It was shown by Garg and Radhakrishnan [8] that for a large constant n_0, for $n \geq n_0$, we have $s_N(m,n,3) \geq \sqrt{mn}$ (In fact, they considered how the non-adaptive complexity depended on the type of query function used; they obtained a lower bound of the form $s_N(m,n,t) = \Omega(m^{1-1/(cn)})$, for a constant c and $n \geq 4$, for all but two of the twenty-two possible classes of functions. Their methods yield a lower bound of $\sqrt{m}$ provided $n \geq 4$.) We show the following.

▶ **Theorem 2** (Result 2). $s_N(m,2,3) = \Omega(\sqrt{m})$

We follow the method of Garg and Radhakrishnan [8], who classified all three input query functions into a small number of equivalence classes and provided either combinatorial or algebraic arguments for each class.

In the rest of this paper, we provide detailed proofs for the above results.

2 Fully-Explicit Adaptive Set Membership scheme

▶ **Theorem 3.** *For every $n \geq 2$ and $t \geq cn\log n$ for some sufficiently large constant c, there exists a fully explicit adaptive $(n, m, (2^t-1)m^{1/(t-\frac{n-1}{2^{t/(2(n-1))}})}, t)$ scheme for the set membership problem.*

We will use letters $x, y, \ldots$ to refer to elements of $[m]$, but view them as strings of length $L = \log m$. We will use the following notation in connection with strings. For a string z, we use $z[i]$ to refer to its i-th component and $z[1,i]$ to refer to the substring $z[1]z[2]\ldots z[i]$. For a sequence of indices $B \subseteq [L]$, we use $x[B]$ to refer to the substring $(x[i] : i \in B)$.

On query "Is $x \in S$?", we will probe locations of our data structure using parts of the bit string x as address. For this, we partition the set of indices into t blocks: $B_1, B_2, \ldots, B_t$, where each B_i is a set/sequence of indices from $[L]$. Notice that the number of blocks is equal to the number of bit probes into the data structure that will be needed to answer membership queries. For $i = 1, 2, \ldots, t-1$ (every block except the last is of equal length), each such block B_i will be further partitioned into t' sub-blocks: $B_{i1}, \ldots B_{it'}$, of equal length. See the example in Figure 1. We denote the length of each of the first $t-1$ blocks by l_b, the length of the final block by l_f, and the length of each sub-block within the first $t-1$ blocks by l_s. The parameters t', l_b, l_s, and l_f will be fixed later.

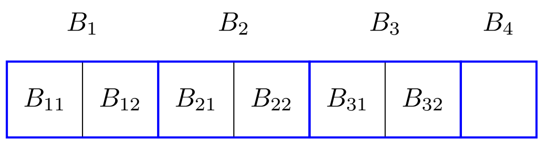

Figure 1 Address of an element divided into blocks (colored blue, $t = 4$, $l_b = 2$, $l_f = 1$) and sub-blocks ($t' = 2$, $l_s = 1$), where the universe is of size 128.

The data structure must be able to distinguish elements in the set from those not in the set. For this, we will identify some blocks and sub-blocks which will be used to differentiate the elements. In the following definition, we assume that the set $S \subseteq [m]$ that we wish to represent in the data structure has been fixed. For $x \in S$, we will designate some blocks as branching blocks. Simultaneously, we will construct and associate a string in $\{0,1\}^{t-1}$ with x, which we refer to as $\mathsf{AuxPath}_x$ (the auxiliary path of x). The strings $\mathsf{AuxPath}_x$ can be thought of as paths leading from a root to a leaf in a binary tree. The pattern of left and right turns that the element x takes down the tree will be closely related to the paths that the corresponding query take in a tree-like data structure.

[Branching block, branching sub-block, AuxPath]

For $i = 1, 2, \ldots, t-1$, we will determine inductively for each $x \in S$ if B_i is a branching block for x and the value of the bit $\mathsf{AuxPath}_x[i]$. We will later extend these definitions to $z \notin S$. Assume $\mathsf{AuxPath}_x[1,i]$ has been determined for all $x \in S$. We say that the block B_i is a *branching block* for $x \in S$ if there exists a $y \in S$, such that

- $\mathsf{AuxPath}_x[1,i-1] = \mathsf{AuxPath}_y[1,i-1]$;
- $x[B_i] \neq y[B_i]$.

If B_i is a branching block for x, then we will designate one of its sub-blocks as a branching sub-block for x. See the example in Figure 2. A sub-block B_{ij} is a branching sub-block of the branching block B_i for x if j is the smallest index (in $[t']$) such that there exists $y \in S$ such that

- $\mathsf{AuxPath}_x[1,i-1] = \mathsf{AuxPath}_y[1,i-1]$;
- $x[B_{ij}] \neq y[B_{ij}]$.

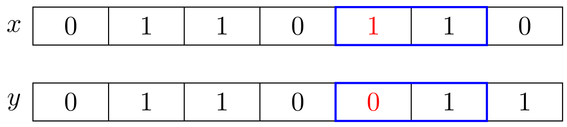

Figure 2 Data structure to store $x = 0110110$ and $y = 0110011$: Branching block (colored blue) is B_3 and branching sub-block (colored red) is B_{31}.

We now define $\mathsf{AuxPath}_x[i]$.

- If B_i is not a branching block for x, then $\mathsf{AuxPath}_x[i] = 0$.
- If B_i is a branching block for x, and $x[B_{ij}]$ is the lexicographically first string in the set

$$\{y[B_{ij}] : y \in S \text{ and } \mathsf{AuxPath}_y[1,i-1] = \mathsf{AuxPath}_x[1,i-1]\},$$

then $\mathsf{AuxPath}_x[i] = 0$, otherwise $\mathsf{AuxPath}_x[i] = 1$.

We now extend the definition of AuxPath to $z \notin S$. Again we define $\mathsf{AuxPath}_z[i]$ inductively for $i = 1, 2, \ldots$. Suppose $\mathsf{AuxPath}_z[1], \ldots, \mathsf{AuxPath}_z[i-1]$ have been determined.

- If $z[B_i] = x[B_i]$ for some some $x \in S$, where $\mathsf{AuxPath}_x[1,i-1] = \mathsf{AuxPath}_z[1,i-1]$, then $\mathsf{AuxPath}_z[i] = \mathsf{AuxPath}_x[i]$.
- If B_i is not a branching block for any $x \in S$, where $\mathsf{AuxPath}_x[1,i-1] = \mathsf{AuxPath}_z[1,i-1]$, then $\mathsf{AuxPath}_z[i] = 1$. (Note that if B_i is a branching block for some $x \in S$ such that $\mathsf{AuxPath}_x[1,i-1] = \mathsf{AuxPath}_z[1,i-1]$, then it is a branching block for all such $x \in S$.)
- Otherwise, let B_{ij} be the branching sub-block of the branching block B_i for some $x \in S$ with $\mathsf{AuxPath}_x[1,i-1] = \mathsf{AuxPath}_z[1,i-1]$. (Note that B_{ij} does not depend on the choice of x.) Then $\mathsf{AuxPath}_z[i] = 1$ iff $z[B_{ij}]$ is the lexicographically first string in $\{y[B_{ij}] : y \in S \text{ and } \mathsf{AuxPath}_y[1,i-1] = \mathsf{AuxPath}_z[1,i-1]\}$. Note that $z \notin S$ are treated differently compared to $x \in S$ here: $\mathsf{AuxPath}_z[i]$ is set 1 precisely when $\mathsf{AuxPath}_x[i]$ would be set to 0 for an $x \in S$ in the same situation. This fact will be exploited later. See Figure 4 for an example.

▶ **Proposition 4.** *There are at most $n-1$ branching blocks and hence at most $n-1$ branching sub-blocks. (Recall that $n = |S|$)*

▶ Remark. Note that $\mathsf{AuxPath}_x[i]$ depends only on $x[B_1 \cup \cdots \cup B_i]$; that is, if $x, y \in S$ and $x[B_1 \cup \cdots \cup B_i] = y[B_1 \cup \cdots \cup B_i]$, then $\mathsf{AuxPath}_x[i] = \mathsf{AuxPath}_y[i]$.

2.1 The storage scheme

The data structure is organized in the form of a binary tree of depth $t-1$. Thus, the tree has t levels of nodes; the root is at level 1 and the leaf is at level t. We will use binary strings as names for the nodes of this binary tree. The root is named by the empty string λ; the left child of the root is named 0 and the right child is named 1. In general, the left child of the node with name σ is named $\sigma 0$, and its right child is named $\sigma 1$. Thus, there is one leaf for each binary string in $\{0,1\}^{t-1}$.

At node σ of the tree we place an array A_σ that can store 2^{l_b} bits. The storage scheme determines the values of A_σ as follows. First, based on the set S, we compute a *message* M of $t-1$ bits. We describe the construction of the string M and its properties below. For each x, we let $\mathsf{TruePath}_x[i] = \mathsf{AuxPath}_x[i] + M[i] \pmod 2$. Now we wish to arrange the contents of the various arrays at levels 1 to $t-1$ in our tree so that $\mathsf{TruePath}_x$ can be read off from their contents. For all x, we set $A_{\mathsf{TruePath}[1,i-1]}[x[B_i]] = \mathsf{TruePath}_x[i]$. Note that the above description assigns values to arrays that reside in our tree at levels 1 to $t-1$. Now, we describe how the bits in A_σ are set, where $\sigma \in \{0,1\}^{t-1}$ (this array resides at level t). For each $x \in S$, let

$$I_x = \{(i,j) : B_{ij} \text{ is a branching sub-block for } x\}.$$

Then, let a_x, the address of the final probe, be obtained by concatenating $x[B_t]$ in the end with the strings $(x[B_{ij}] : (i,j) \in I_x)$ (note that $x[B_t]$ has not been used in the previous probes), adding 0's at the end if necessary so that a_x has exactly l_b bits. Then, set $A_{\mathsf{TruePath}_x[1,t-1]}[a_x] = 1$. Finally, set all bits whose values are not specified above to 0. An example is illustrated in Figure 3. To complete the description of the storage scheme, we need to specify how the string $M \in \{0,1\}^{t-1}$ is obtained and fix the parameters l_b, l_f, l_s and t'.

The message M

When we answer a query, the message M will help us identify the branching sub-blocks on the query element's auxiliary path. Note that the true path and the message differ in at most $n-1$ positions. Thus, if M can be recovered even when up to $n-1$ of its bits are flipped, then we can extract M from the path (i.e., $\mathsf{TruePath}_x$) the element takes down the tree. The values in the arrays are so arranged that they simultaneously achieve two goals: (i) they ensure elements in the set and those outside are sent to different leaves; (ii) if despite (i) an element in the set and an element outside reach the same leaf, the path allows us to extract the message M, using which we ensure that these two elements read different bits on their last probe.

As observed above, the total number of branching sub-blocks is at most $n-1$. The address of all these sub-blocks put together will be encoded using an error-correcting code of length $t-1$. The resulting codeword will be the string M. The main property of this codeword that we need now is that it is has a distance greater than $2(n-1)$, so that up to $n-1$ errors can be corrected.

Fixing the parameters of the data-structure

Since there are t' sub-blocks within each block, we have

$$t' \cdot l_s = l_b. \tag{4}$$

The length of the final address is $(n-1)l_s + l_f$. Since the array at each node stores 2^{l_b} bits, we have

$$l_b = (n-1)l_s + l_f. \tag{5}$$

Finally, since the entire address of $x \in [m]$ is partitioned into blocks $B_1, B_2, \ldots, B_t$, we have

$$(t-1)l_b + l_f = L. \tag{6}$$

Solving Equations (4) to (6) while fixing t and t', we get $l_b = \frac{L}{t-\frac{n-1}{t'}}$, $l_f = L - \frac{L(t-1)}{t-\frac{n-1}{t'}}$ and $l_s = \frac{L}{tt'-(n-1)}$. Therefore, the size of the data structure that we constructed is $(2^t - 1)m^{1/(t-(n-1)/2^{t'})}$. Observe that, asymptotically, the size of the data-structure increases as we increase t'. Later, we will show that for the existence of error-correcting codes with properties described in 2.1, we need $t' \geq t/(2(n-1))$. Therefore, the optimal choice of t' that minimizes the size of the data-structure is $t' = t/(2(n-1))$.

2.2 The query scheme

In this section, we describe how given an element $x \in [m]$, we can make t probes to the data structure and determine with certainty if $x \in S$. The first $t-1$ probes are easy to anticipate based on the description of the storage scheme above: Having read $b_1, b_2, \ldots, b_{i-1}$ in the first $i-1$ probes, we set the i-th bit $b_i = A_{b[1,i-1]}[x[B_i]]$, probing the data structure once more. After $t-1$ such probes have been made, the query scheme obtains $b[1,t-1]$. Note that this string corresponds to $\mathsf{TruePath}[x]$ defined above. In some cases, we will be able to answer the query based on just $b[1,i-1]$. In other cases, the final probe will be made, as expected, into the array $A_{b[1,t-1]}$. Some care is needed in determining the location. The bit read in this location will be the answer to the query "Is x in S?".

If $b[1,t-1]$ is *not* within distance $n-1$ of any codeword, then we declare that x is not in S. Else, we decode $b[1,t-1]$ to obtain M. Note that for $x \in S$, $\mathsf{TruePath}_x[i] \neq M[i]$ iff $\mathsf{AuxPath}_x[i] = 1$ (and B_i is a branching block). Therefore, $\Delta(\mathsf{TruePath}[x], M) \leq n-1$, where $\Delta(\cdot)$ is the Hamming distance function. Since the code we chose has distance at least $2(n-1)$, we can decode $\mathsf{TruePath}[x]$ to obtain M and hence, all the branching sub-blocks, say, $B_{i_1j_1}, B_{i_2j_2}, \ldots, B_{i_rj_r}$, where r is at most $n-1$. The location for the final probe will be obtained by concatenating these strings as follows: $x[B_{i_1j_1}]\cdots x[B_{i_rj_r}]x[B_t]$.

2.3 Proof of correctness

In this section, we show that the above scheme answers every query correctly: for every element $z \in [m]$, we show that the bit read in the final probe is 1 iff $z \in S$. Let a_z denote the address of the final probe corresponding to z.

For $x \in S$, our storage scheme ensures that $A_{\mathsf{TruePath}_x[1,t-1]}[a_x] = 1$. Now, fix $z \notin S$. From the definition of the storage scheme, it follows that if $A_{\mathsf{TruePath}_z[1,t-1]}[a_z] = 1$, then $\exists x \in S$ such that $\mathsf{TruePath}_z[1,t-1] = \mathsf{TruePath}_x[1,t-1]$ and $a_z = a_x$. Fix x such that $\mathsf{TruePath}_z[1,t-1] = \mathsf{TruePath}_x[1,t-1]$; we will argue that for $a_z \neq a_x$. Let $i \in [t]$ such that $z[B_i] \neq x[B_i]$. If $i = t$, then clearly $a_z \neq a_x$, and we are done. Hence assume that $i < t$. Since $\mathsf{TruePath}_z[1,t-1] = \mathsf{TruePath}_x[1,t-1]$, we have $\mathsf{AuxPath}_z[1,t-1] = \mathsf{AuxPath}_x[1,t-1]$. In particular, since $\mathsf{AuxPath}_z[1,i-1] = \mathsf{AuxPath}_x[1,i-1]$ and $\mathsf{AuxPath}_z[i] = \mathsf{AuxPath}_x[i]$, it follows from the definition of the storage scheme that B_i is a branching block. Let B_{ij} be the branching sub-block of B_i. Observe that $\mathsf{AuxPath}_z[i] = 1$ iff $z[B_{ij}]$ is the lexicographically first string in $\{y[B_{ij}] : y \in S \text{ and } \mathsf{AuxPath}_y[1,i-1] = \mathsf{AuxPath}_z[1,i-1]\}$. On the other hand, $\mathsf{AuxPath}_x[i] = 1$ iff $x[B_{ij}]$ is the NOT the lexicographically first string in $\{y[B_{ij}] : y \in S$ and $\mathsf{AuxPath}_y[1, i-1] = \mathsf{AuxPath}_x[1, i-1]\}$. Therefore, $\mathsf{AuxPath}_z[i] = \mathsf{AuxPath}_x[i]$ only if $z[B_{ij}] \neq x[B_{ij}]$. Since $z[B_{ij}]$ is a substring of a_z and $x[B_{ij}]$ is a substring of a_x (they occupy identical locations), it follows that $a_z \neq a_x$. See Figure 4 for an example.

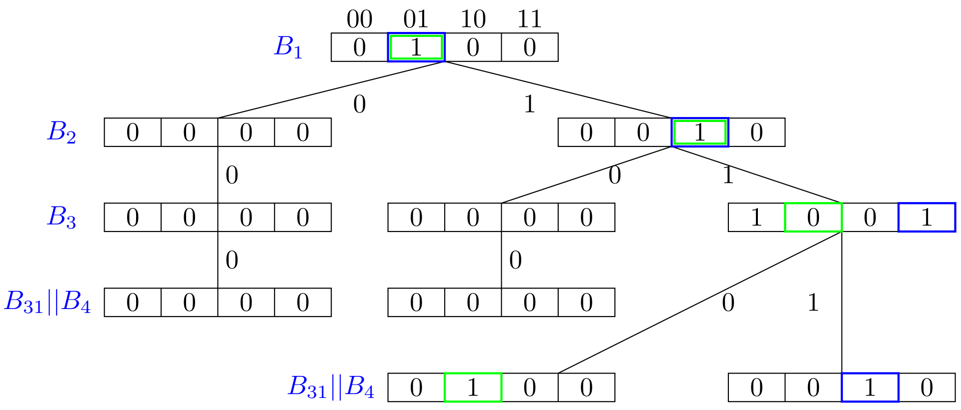

Figure 3 Data structure to store $x = 0110110$ and $y = 0110011$: $\mathsf{AuxPath}_x = 001$ and $\mathsf{AuxPath}_y = 000$ and $M = 110$ (not encoded in this example, for simplicity), where 11 indicates the branching block and 0 is the branching sub-block; $\mathsf{TruePath}_x = 111$ (colored blue) and $\mathsf{TruePath}_y = 110$ (colored green); last bit read is 1.

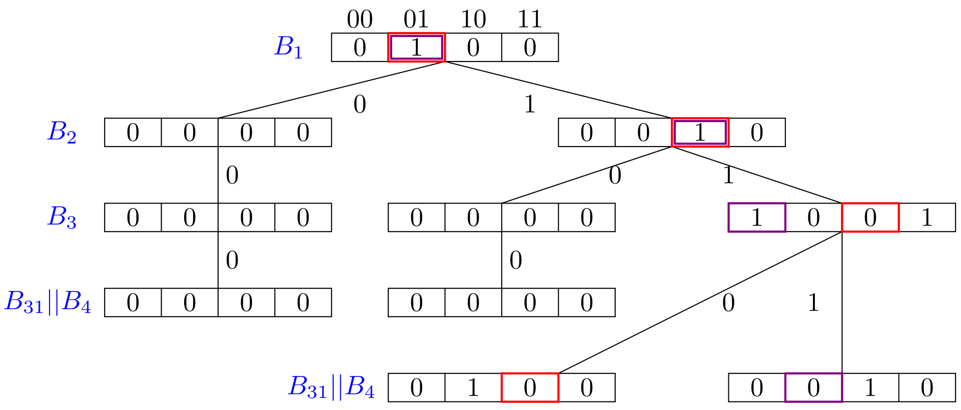

Figure 4 Verifying correctness: paths taken by elements which *resemble* x and y, i.e., differ only in one bit and agree on the branching sub-block B_{31}; $z_1 = 0110100$ (colored red) and $z_2 = 0110001$ (colored violet); the last bit read is 0.

Existence of error-correcting codes

Let $\mathcal{C}[N, K, D]$ denote a binary code of block length N, dimension K, and distance D. The following proposition gives sufficient conditions for the existence of such a code.

▶ **Proposition 5.** *For $N, k, D \in \mathbb{N}$ such that $2^N \geq 2^K \binom{N}{D-1} D$ and $N > 2(D-1)$, there exists a $\mathcal{C}[N, K, D]$ code.*

Proof. Consider the vector space $\mathbb{F}_2^N$. The volume of a Hamming ball of radius $D-1$ centered at any point in $\mathbb{F}_2^N$ is $\sum_{i=0}^{D-1} \binom{N}{i}$. When $N > 2(D-1)$, this volume is at most $2\binom{N}{D-1}D$. Since $2^N \geq 2^K \binom{N}{D-1} D$, we can greedily pick a code $\mathcal{C} \subseteq \mathbb{F}_2^N$ of size 2^K such that the distance between any two codewords in $\mathcal{C}$ is at least D. ◀

The message that we want to encode consists of the addresses of all branching sub-blocks, padded with 0's in the end, if necessary, to maintain uniform length. Let K denote the length of this message to be encoded. Since each sub-block is indexed by $\log(t-1)+\log t'$ bits and there are at most $n-1$ branching sub-blocks, we take $K=(n-1)[\log(t-1)+\log t']$. The desired length of the codeword is $N=t-1$ (equal to the number of bits read until the final probe) and we need the distance $D \geq 2(n-1)+1$. It follows from Proposition 5 that for $t'=2^{t/(2(n-1))}$ and $t \geq cn\log n$, for a sufficiently large constant c, there exists a $\mathcal{C}[t-1,(n-1)[\log(t-1)+\log t'],2(n-1)+1]$ code.

Note that we may also explicitly construct the desired error-correcting code as a linear BCH code [3, 11], which is an $[N,K,D]$ code with $K \geq N-\frac{1}{2}(D-1)\log(N+1)$ (this bound appears, e.g., in the lecture notes of Guruswami [10, Notes 6, Lemma 5]). Substituting $N=t-1$ and $D=2(n-1)+1$, we obtain a code with message length

$$K \geq (t-1)-(n-1)\log t \geq (n-1)[\log(t-1)+\log t'],$$

for $t \geq cn\log n$, for a sufficiently large constant c. This explicit use of BCH codes will enable us to use their efficient encoding and decoding algorithms [3, 11]. In particular, the running time of these algorithms is bounded by a polynomial in n and t. In our application, we have $n,t \ll \log m$; so the running time of the query algorithm is bounded by a polynomial in $\log m$.

3 Non-adaptive Set Membership

In this section, we discuss our result on deterministic query schemes with three non-adaptive probes. Theorem 8 is an improvement of the lower bound proved by Garg and Radhakrishnan [8].

In principle, there can be different query functions for different elements. But since there are only a finite number (256) of boolean functions on three variables, by the pigeon-hole principle, some set of at least m/256 elements of the universe use a common query function. We may thus restrict our attention to this part of the universe, and assume that the query function is the same for every element.

▶ **Definition 6** (Equivalence). *A boolean function $f(x_1,x_2,\ldots,x_k):\{0,1\}^k \to \{0,1\}$ is said to be equivalent to a boolean function $g(x_1,x_2,\ldots,x_k):\{0,1\}^k \to \{0,1\}$ if f can be obtained from g through a sequence of negations and permutations of the variables in g.*

▶ **Proposition 7.** *Let $f,g:\{0,1\}^k \to \{0,1\}$ be equivalent. If s_1 and s_2 are the minimum bits of space required for non-adaptive (m,n,s_1,t) and (m,n,s_2,t)-schemes with query functions f and g respectively, then $s_1=s_2$.*

For three-variable boolean functions, Polýa counting yields that there are twenty-two equivalence classes [24, 20]; they are listed explicitly on this page [19]. We prove Theorem 8 for each of these equivalence classes separately. We omit the proofs for certain classes of functions when the argument is essentially the one given by Garg and Radhakrishnan [7] and refer the reader to that paper for more details.

▶ **Theorem 8.** *$s_N(m,n=2,t=3)=\Omega(\sqrt{m})$.*

Proof. We focus on the function that the query algorithm applies to the three bits read to answer the query. Since there are only a finite number of such three-variable boolean functions, we may restrict ourselves to a constant fraction of the universe for which the query function is the same, which we denote by f. We also assume that the memory consists of three arrays A_1, A_2, A_3, each of size s, and each probe is made on a different array. We will show that $s \geq \sqrt{m}/3$. Let us assume to the contrary that $s < \sqrt{m}/3$. To arrive at a contradiction, we will show that there exists a set $S \subseteq [m]$ of size at most 2 that cannot be stored in the data structure. We need to cover all possible query functions f. As observed by Garg and Radhakrishnan [7], these functions fall into a small number of equivalence classes. For most classes, their arguments already yield the result.

If there is a polynomial (which we may assume is multilinear) of degree at most two (in variables (x, y, z) over any field) that is 0 whenever the function f evaluates to 0, then there will be a set of size at most 2 that cannot be stored in the data structure [21]. This covers the following functions (the first column lists a representative from the equivalence class). In the second column, the polynomials listed are over the field $\mathbb{F}_2$ except when the query function is $x + y + z = 1$.

Table 1 Polynomial representation of query functions.

Query function	**Polynomial**
CONSTANT	$0, 1$
x	x
$x + y + z \neq 1$	$x + y + z - 1$
$x + y + z = 1$	$x + y + z + xy + yz + zx$ (over $\mathbb{F}_3$)
$x \wedge y$	xy
$\neg x \vee \neg y$	$1 + xy$
$(x \wedge y) \vee (\neg x \wedge z)$	$z + xy + xz$
$(x \wedge y) \vee (\neg x \wedge \neg y)$	$1 + x + y$
MAJORITY(x, y, z)	$xy + yz + zx$
PARITY(x, y, z)	$x + y + z$
$(x \wedge y) \oplus z$	$z + xy$
$(x \oplus y) \wedge z$	$yz + zx$
$\neg[(x \oplus y) \wedge z]$	$1 + yz + zx$
ALL-EQUAL(x, y, z)	$1 + x + y + z + xy + yz + zx$
NOT ALL-EQUAL(x, y, z)	$x + y + z + xy + yz + zx$

The remaining query functions are equivalent to one of the following six functions.

- $f(x, y, z) = (x \vee y) \wedge z$ or its complement.
- $f(x, y, z) = x \wedge y \wedge z$ or its complement.
- $f(x, y, z) = (x \wedge y \wedge z) \vee (\neg y \wedge \neg z)$ or its complement.

Using the fact that dense graphs have short cycles, Garg and Radhakrishnan show that when the query function is equivalent to $f(x, y, z) = (x \wedge y \wedge z) \vee (\neg y \wedge \neg z)$, the size of the data structure is at least $(1/7)m^{1-1/(\lfloor n/4 \rfloor+1)}$, which implies a lower bound of $\Omega(\sqrt{m})$ only if $n \geq 4$. For the functions equivalent to $f(x, y, z) = (x \vee y) \wedge z$, $f(x, y, z) = (x \wedge y \wedge z)$ and their complements, they proved a linear lower bound for $n \geq 3$. Their proof crucially relied on the notion of a *private vertex*: a private vertex for an element $u \in [m]$ is a memory location that is probed only for the query corresponding to u and for no other query. These arguments do not seem to be directly applicable when $n = 2$. We define a closely related notion of *private edge* and use it to prove an $\Omega(\sqrt{m})$ lower bound for the remaining six classes of functions mentioned above.

We say that an element u has a private edge if $|\{l_1(u), l_2(u), l_3(u)\} \cap \{l_1(v), l_2(v), l_3(v)\}| \leq 1$ for all $v \in [m] \setminus \{u\}$; that is, no pair of locations probed for query element u is probed also for a different query element v. At most $3s^2$ elements have private edges. Since $s \leq \sqrt{m}/3$, we infer that $\mathcal{U} = \{u \in [m] : u \text{ has no private edge}\}$ has at least $2m/3$ elements.

Now suppose the query function is $f(x, y, z) = (x \wedge y \wedge z) \vee (\neg y \wedge \neg z)$. Fix distinct $u, v \in \mathcal{U}$ such that $(l_2(u), l_3(u)) = (l_2(v), l_3(v))$ (then $l_1(u) \neq l_1(v)$). Since v has no private edge, there exists $w \in [m] \setminus \{u, v\}$ such that $(l_1(v), l_3(v)) = (l_1(w), l_3(w))$. That is, u and v make their second and third probes to identical locations, and v and w make their first and third probes to identical locations. We claim that any assignment under which the queries for both u and w evaluate to 1, the query for the element v also evaluates to 1; thus the set $\{u, w\}$ cannot be stored by this data structure. Since $u \in S$ and $v \notin S$, the nature of our function implies that $A_1[l_1(u)], A_2[l_2(u)], A_3[l_3(u)] = 1$, implying that $A_2[l_2(v)], A_3[l_3(v)] = 1$. But then, $A_3[l_3(w)] = 1$, and again the form of our function implies that $A_1[l_1(w)] = 1$. But $l_1(w) = l_1(v)$; so all together we have that $A_1[l_1(v)], A_2[l_2(v)], A_3[l_3(v)] = 1$, but then the query for v will return 1, as claimed. For the complement of $f(x, y, z) = (x \vee y \vee z) \wedge (\neg y \vee \neg z)$, note that an analogous argument shows that whenever the query function returns 0 for u and w then it returns 0 on v as well, that is, no such scheme can store the set $\{v\}$ – a contradiction.

Next, suppose the query function is $f(x, y, z) = (x \vee y) \wedge z$. Consider an element $u \in \mathcal{U}$. Since u does not have a private edge, there exist distinct elements $v, w \in [m] \setminus \{u\}$ such that $(l_1(u), l_3(u)) = (l_1(v), l_3(v))$ and $(l_2(u), l_3(u)) = (l_2(w), l_3(w))$. Let $S = \{u\}$ be the set stored in the data structure. Since $u \in S$, it must be the case that $A_3[l_3(u)] = 1$ and, $A_1[l_1(u)] = 1$ or $A_2[l_2(u)] = 1$. It follows that $A_3[l_3(v)] = A_3[l_3(w)] = 1$ and, $A_1[l_1(v)] = 1$ or $A_2[l_2(w)] = 1$ and hence the query function returns value 1 for at least one of v and w; but neither of them is in S – a contradiction. The argument for the complement of this function is analogous: the set $\{v, w\}$ cannot be stored.

Finally, suppose the query function is $f(x, y, z) = (x \wedge y \wedge z)$. There exist two distinct elements $u, v \in \mathcal{U}$ such that $(l_2(u), l_3(u)) = (l_2(v), l_3(v))$. Since v has no private edge, there exists $w \in [m] \setminus \{u, v\}$ such that $(l_1(v), l_3(v)) = (l_1(w), l_3(w))$. Let $S = \{u, w\}$ be the set stored in the data structure. Then, $A_1[l_1(v)], A_2[l_2(v)], A_3[l_3(v)] = 1$. But then the query function for v will also return 1, but v is not in S – a contradiction. The argument for the complement of this function is analogous: the set $S = \{v\}$ cannot be stored.

Thus, we have an $\Omega(\sqrt{m})$ lower bound on $s_N(m, 2, 3)$ for all query functions. ◀

References

1 Mirza Galib Anwarul Husain Baig and Deepanjan Kesh. Two new schemes in the bitprobe model. In M. Sohel Rahman, Wing-Kin Sung, and Ryuhei Uehara, editors, *WALCOM: Algorithms and Computation - 12th International Conference, WALCOM 2018, Dhaka, Bangladesh, March 3-5, 2018, Proceedings*, volume 10755 of *Lecture Notes in Computer Science*, pages 68–79. Springer, 2018. doi:10.1007/978-3-319-75172-6_7.

2 Burton H. Bloom. Space/time trade-offs in hash coding with allowable errors. *Communications of the ACM*, 1970.

3 R.C. Bose and D.K. Ray-Chaudhuri. On a class of error correcting binary group codes. *Information and Control*, 3(1):68–79, 1960. doi:10.1016/S0019-9958(60)90287-4.

4 Andrej Brodnik and J. Ian Munro. Membership in constant time and almost-minimum space. *SIAM J. Comput.*, 28(5):1627–1640, May 1999. doi:10.1137/S0097539795294165.

5 H. Buhrman, P. B. Miltersen, J. Radhakrishnan, and S. Venkatesh. Are bitvectors optimal? *SIAM J. Comput.*, 31(6):1723–1744, June 2002. doi:10.1137/S0097539702405292.

6 Michael L. Fredman, János Komlós, and Endre Szemerédi. Storing a sparse table with 0(1) worst case access time. *J. ACM*, 31(3):538–544, June 1984. doi:10.1145/828.1884.

7 Mohit Garg and Jaikumar Radhakrishnan. Set membership with non-adaptive bit probes. *CoRR*, abs/1612.09388, 2016. arXiv:1612.09388.

8 Mohit Garg and Jaikumar Radhakrishnan. Set membership with non-adaptive bit probes. In *Proc. 34th Symposium on Theoretical Aspects of Computer Science, STACS 2017, March 8-11, 2017, Hannover, Germany*, pages 38:1–38:13, 2017. doi:10.4230/LIPIcs.STACS.2017.38.

9 Mohit Garg and Jaikumar Radhakrishnan. Set membership with a few bit probes. In *Proc. 26th Annual ACM-SIAM Symposium on Discrete Algorithms, SODA*, pages 776–784, San Diego, CA, USA, January 4-6, 2015. doi:10.1137/1.9781611973730.53.

10 Venkatesan Guruswami. 15-859v: Introduction to coding theory, Spring 2010, CMU, 2010. URL: http://www.cs.cmu.edu/~venkatg/teaching/codingtheory/notes/notes6.pdf.

11 Alexis Hocquenghem. Codes correcteurs d'erreurs. *Chiffres (Paris)*, 2:147–156, 1959.

12 Deepanjan Kesh. Space Complexity of Two Adaptive Bitprobe Schemes Storing Three Elements. In Sumit Ganguly and Paritosh Pandya, editors, *38th IARCS Annual Conference on Foundations of Software Technology and Theoretical Computer Science (FSTTCS 2018)*, volume 122 of *Leibniz International Proceedings in Informatics (LIPIcs)*, pages 12:1–12:12, Dagstuhl, Germany, 2018. Schloss Dagstuhl–Leibniz-Zentrum fuer Informatik. doi:10.4230/LIPIcs.FSTTCS.2018.12.

13 Moshe Lewenstein, J. Ian Munro, Patrick K. Nicholson, and Venkatesh Raman. Improved explicit data structures in the bitprobe model. In *Proc. 22nd Annual European Symposium on Algorithms*, pages 630–641, Wroclaw, Poland, September 8-10, 2014. doi:10.1007/978-3-662-44777-2_52.

14 A. Makhdoumi, S. Huang, M. Médard, and Y. Polyanskiy. On locally decodable source coding. In *2015 IEEE International Conference on Communications (ICC)*, pages 4394–4399, 2015.

15 Marvin Minsky and Seymour A. Papert. *Perceptrons: An Introduction to Computational Geometry*. The MIT Press, 2017.

16 Patrick K. Nicholson, Venkatesh Raman, and S. Srinivasa Rao. *A Survey of Data Structures in the Bitprobe Model*, pages 303–318. Springer Berlin Heidelberg, Berlin, Heidelberg, 2013. doi:10.1007/978-3-642-40273-9_19.

17 Rasmus Pagh. Low redundancy in static dictionaries with constant query time. *SIAM J. Comput.*, 31(2):353–363, February 2002. doi:10.1137/S0097539700369909.

18 Mihai Patrascu. Succincter. In *Proceedings of the 2008 49th Annual IEEE Symposium on Foundations of Computer Science*, FOCS '08, page 305–313, USA, 2008. IEEE Computer Society. doi:10.1109/FOCS.2008.83.

19 Tilman Piesk. The 22 becs, 3-ary boolean functions. *Wikiversity*, 2016. URL: https://en.wikiversity.org/w/index.php?title=3-ary_Boolean_functions&oldid=1587287.

20 G. Pólya. Kombinatorische anzahlbestimmungen für gruppen, graphen und chemische verbindungen. *Acta Math.*, 68:145–254, 1937. doi:10.1007/BF02546665.

21 J. Radhakrishnan, P. Sen, and S. Venkatesh. The quantum complexity of set membership. In *Proceedings of the 41st Annual Symposium on Foundations of Computer Science*, FOCS '00, page 554, USA, 2000. IEEE Computer Society.

22 Jaikumar Radhakrishnan, Venkatesh Raman, and S. Srinivasa Rao. Explicit deterministic constructions for membership in the bitprobe model. In *Proc. 9th Annual European Symposium on Algorithms Algorithms*, pages 290–299, Aarhus, Denmark, August 28-31, 2001. doi:10.1007/3-540-44676-1_24.

23 Jaikumar Radhakrishnan, Smit Shah, and Saswata Shannigrahi. Data structures for storing small sets in the bitprobe model. In *Proc. 18th Annual European Symposium on Algorithms Algorithms*, pages 159–170, Liverpool, UK, September 6-8, 2010. doi:10.1007/978-3-642-15781-3_14.

24 J. Howard Redfield. The theory of group-reduced distributions. *American Journal of Mathematics*, 49(3):433–455, 1927. URL: http://www.jstor.org/stable/2370675.