A locally encodable and decodable compressed data structure

The MIT Faculty has made this article openly available. Please share how this access benefits you. Your story matters.

<table>
<thead>
<tr>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>As Published</td>
<td><a href="http://dx.doi.org/10.1109/ALLERTON.2009.5394919">http://dx.doi.org/10.1109/ALLERTON.2009.5394919</a></td>
</tr>
<tr>
<td>Publisher</td>
<td>Institute of Electrical and Electronics Engineers</td>
</tr>
<tr>
<td>Version</td>
<td>Final published version</td>
</tr>
<tr>
<td>Accessed</td>
<td>Wed Mar 13 08:46:02 EDT 2019</td>
</tr>
<tr>
<td>Citable Link</td>
<td><a href="http://hdl.handle.net/1721.1/58949">http://hdl.handle.net/1721.1/58949</a></td>
</tr>
<tr>
<td>Terms of Use</td>
<td>Article is made available in accordance with the publisher's policy and may be subject to US copyright law. Please refer to the publisher's site for terms of use.</td>
</tr>
<tr>
<td>Detailed Terms</td>
<td></td>
</tr>
</tbody>
</table>
A Locally Encodable and Decodable Compressed Data Structure

Venkat Chandar Devavrat Shah Gregory W. Wornell

Abstract—In a variety of applications, ranging from high-speed networks to massive databases, there is a need to maintain histograms and other statistics in a streaming manner. Motivated by such applications, we establish the existence of efficient source codes that are both locally encodable and locally decodable. Our solution is an explicit construction in the form of a (randomized) data structure for storing $N$ integers. The construction uses multi-layered sparse graph codes based on Ramanujan graphs, and has the following properties: (a) the structure utilizes minimal possible space, and (b) the value of any of the integers can be read or updated in near constant time (on average and with high probability). By contrast, data structures proposed in the context of streaming algorithms and compressed sensing in recent years (e.g., various sketches) support local encodability, but not local decodability; and those known as succinct data structures are locally decodable, but not locally encodable.

I. INTRODUCTION

In this paper, we are concerned with the designing space efficient and dynamic data structures for maintaining a collection of integers, motivated by many applications that arise in networks, databases, signal processing, etc. For example, in a high-speed network router one wishes to count the number of packets per flow in order to police the network or identify a malicious flow. As another example, in a very large database, relatively limited main memory requires “streaming” computation of queries. And for “how many?” type queries, this leads to the requirement of maintaining dynamically changing integers.

In these applications, there are two key requirements. First, we need to maintain such statistics in a dynamic manner so that they can be updated (increment, decrement) and evaluated very quickly. Second, we need the data to be represented as compactly as possible to minimize storage, which is often highly constrained in these scenarios. Therefore, we are interested in data structures for storing $N$ integers in a compressed manner and be able to read, write/update any of these $N$ integers in near constant time.2

A. Prior work

Efficient storage of a sequence of integers in a compressed manner is a classical problem of source coding, with a long history, starting with the pioneering work by Shannon [13].

The work of Lempel and Ziv [14] provided an elegant universal compression scheme. Since that time, there has continued to be important progress on compression algorithms. However, most well-known schemes suffer from two drawbacks. First, these schemes do not support efficient updating of the compressed data. This is because in typical systems, a small change in one value in the input sequence leads to a huge change in the compressed output. Second, such schemes require the whole sequence to be decoded from its compressed representation in order to recover even a single element of the input sequence.

A long line of research has addressed the second drawback. For example, Bloom filters [35], [36], [37] are a popular data structure for storing a set in compressed form while allowing membership queries to be answered in constant time. The rank/select problem [27], [28], [29], [30], [31], [32], [33], [34] and dictionary problem [38], [39], [32] in the field of succinct data structures are also examples of problems involving both compression and the ability to efficiently recover a single element of the input, and [26] gives a succinct data structure for arithmetic coding that supports efficient, local decoding but not local encoding. In summary, this line of research successfully addresses the second drawback but not the first drawback, e.g., changing a single value in the input sequence can still lead to huge changes in the compressed output.

Similarly, several recent papers have examined the first drawback but not the second. In order to be able to update an individual integer efficiently, one must consider compression schemes that possess some kind of continuity property, whereby a change in a single value in the input sequence leads to a small change in the compressed representation. In recent work, Varshney et al. [40] analyzed “continuous” source codes from an information theoretic point of view, but the notion of continuity considered is much stronger than the notion we are interested in, and [40] does not take computational complexity into account. Also, Mossel and Montanari [1] have constructed space efficient “continuous” source codes based on nonlinear, sparse graph codes. However, because of the nonlinearity, it is unclear that the continuity property of the code is sufficient to give a computationally efficient algorithm for updating a single value in the input sequence.

In contrast, linear, sparse graph codes have a natural, computationally efficient algorithm for updates. A large body of work has focussed on such codes in the context of streaming algorithms and compressed sensing. Most of this work essentially involves the design of sketches based on linear codes [4], [5], [6], [3], [2], [7], [8], [9], [10], [11]. Among these, [3], [2], [7], [8], [9], [10], [11] consider sparse linear codes.
Existing solutions from this body of work are ill-suited for our purposes for two reasons. First, the decoding algorithms (LP based or combinatorial) require decoding all integers to read even one integer. Second, they are sub-optimal in terms of space utilization when the input is very sparse.

Perhaps the work most closely related to this paper is that of Lu et al. [21], which develops a space efficient linear code. They introduced a multi-layered linear graph code (which they refer to as a Counter Braid). In terms of graph structure, the codes considered in [21] are essentially identical to Tornado codes [41], so Counter Braids can be viewed as one possible generalization of Tornado codes designed to operate over the integers instead of a finite field. [21] establishes the space efficiency of counter braids when (a) the graph structure and layers are carefully chosen depending upon the distributional information of inputs, and (b) the decoding algorithm is based on maximum likelihood (ML) decoding, which is in general computationally very expensive. The authors propose a message-passing algorithm to overcome this difficulty and provide an asymptotic analysis to quantify the performance of this algorithm in terms of space utilization. However, the message-passing algorithm may not provide optimal performance in general. In summary, the solution of [21] does not solve our problem of interest because: (a) it is not locally decodable, i.e., it does not support efficient evaluation of a single input\(^3\), (b) the space efficiency requires distributional knowledge of the input, and (c) there is no explicit guarantee that the space requirements for the data structure itself (separate from its content) are not excessive; the latter is only assumed.

B. Our contribution

As the main result of this paper, we provide a space efficient data structure that is locally encodable and decodable. Specifically, we consider the following setup.

**Setup.** Let there be \( N \) integers, \( x_1, \ldots, x_N \) with notation \( x = [x_i]_{1 \leq i \leq N} \). Define

\[
S(B_1, B_2, N) = \{x \in \mathbb{N}^N : \|x\|_1 \leq B_1, \|x\|_\infty \leq B_2\}.
\]

Let \( B_2 \geq B_1 \) without loss of generality, where \( B_1, B_2 \) can be arbitrary functions of \( N \) with \( B_2 \leq NB_1 \). Of particular interest is the regime in which both \( N \) and \( B_1 \) (and thus \( B_2 \)) are large. Given this, we wish to design a data structure that can store any \( x \in S(B_1, B_2, N) \) that has the following desirable properties:

- **Read.** For any \( i, 1 \leq i \leq N, x_i \) should be read in near constant time. That is, the data structure should be locally decodable.
- **Write/Update.** For any \( i, 1 \leq i \leq N, x_i \) should be updated (increment, decrement or write) in near constant time. That is, the data structure is locally encodable.
- **Space.** The amount of space utilized is minimal possible. The space utilized for storing auxiliary information about the data structure (e.g. pointers or random bits) as well as the space utilized by the read and write/update algorithms should be accounted for.

**Naive Solutions Don’t Work.** We briefly discuss two naive solutions, each with some of these properties but not all, that will help explain the non-triviality of this seemingly simple question. First, if each integer is allocated \( \log B_2 \) bits, then one solution is the standard Random Access Memory setup: it has \( O(1) \) read and write/update complexity. However, the space required is \( N \log B_2 \), which can be much larger than the minimal space required (as we’ll establish, it’s essentially \( N \log B_1 \)). To overcome this poor space utilization, consider the following “prefix free” coding solution. Here, each \( x_i \) is stored in a serial manner: each of them utilizes (roughly) \( \log x_i \) bits to store its value, and these values are separated by (roughly) \( \log \log x_i \) Is followed by a 0. Such a coding will be essentially optimal as it will utilize (roughly) \( N(\log(B_1 + \log \log B_1)) \) bits. However, read and write/update will require \( \Omega(N) \) operations. As noted earlier, the streaming data structures and succinct data structures have either good encoding or good decoding performance along with small space utilization, but not both. In this paper we desire a solution with both good encoding and decoding performance with small space.

**Our Result.** We establish the following result as the main contribution of this paper.

**Theorem I.1** The data structure described in Section II achieves the following:

- **Space efficiency.** The total space utilization is \( (1 + o(1))N \log B_1^4 \). This accounts for space utilized by auxiliary information as well.
- **Local encoding.** The data structure is based on a layered, sparse, linear graphical code with an encoding algorithm that takes \( \text{poly}(\max(\log(B_2/B_1^2), 1)) \) time to write/update.
- **Local decoding.** Our local decoding algorithm takes

\[
\exp\left(O(\max(1, \log \log(B_2/B_1^2) \times \log \log(B_2/B_1^2)))\right) = \text{quasipoly}(\max(\log(B_2/B_1^2), 1))
\]

on average and w.h.p.\(^4\) to read \( x_i \) for any \( i \). The algorithm produces correct output w.h.p.. Note that the randomness is in the construction of the data structure.

In summary, our data structure is optimal in terms of space utilization as a simple counting argument for the size of \( S(B_1, B_2, N) \) suggests that the space required is at least \( N \log B_1 \); encoding/decoding can take \( \text{poly}(\log N) \) time at worst, e.g. when \( B_1 = O(1) \) and \( B_2 = N^\alpha \) for some \( \alpha \in (0, 1) \), but in more balanced situations, e.g. when \( B_2 = N^\alpha \) and \( B_1 = N^{\alpha/2} \), the encoding/decoding time can be \( O(1) \).

In what follows, we describe a construction with the properties stated in Theorem I.1, but, due to space constraints, we omit all proofs. Before detailing our construction, we provide

\[^3\]For example, see the remark in Section 1.2 of [21].

\[^4\]The \( o(1) \) here is with respect to both \( B_1 \) and \( N \), i.e., the \( o(1) \to 0 \) as \( \min(N, B_1) \to \infty \).

\[^5\]In this paper, unless stated otherwise by with high probability (w.h.p.) we mean probability \( 1 - 1/\text{poly}(N) \).
some perspectives. The starting point for our construction is the set of layered, sparse, linear graphical codes introduced in [21]. In order to obtain the desired result (as explained above), we have to overcome two non-trivial challenges. The first challenge concerns controlling the space utilized by the auxiliary information in order to achieve overall space efficiency. In the context of linear graphical codes, this includes information about adjacency matrix of the graph. It may require too much space to store the whole adjacency matrix explicitly (at least $\Omega(N \log N)$ space for an $N \times \Theta(N)$ bipartite graph). Therefore, it is necessary to use graphs that can be stored succinctly. The second challenge concerns the design of local encoding and decoding algorithms. Sparse linear codes readily allow such succinct representation. But the design of local decoding algorithm is not clear. All the known decoding algorithms in the literature are designed for decoding all the information simultaneously. Therefore, we need to come up with novel decoding algorithm.

To address the first challenge, we use pseudorandom graphs based on appropriate Ramanujan graphs. These graphs have efficient ‘look up’, a succinct description (i.e., $O(\sqrt{N} \log N)$ space), and a ‘locally tree-like’ property (i.e., girth at least $\Omega(\log N)$). To address the second challenge, using this locally tree-like property and the classical message-passing algorithm, we design a local decoding algorithm that is essentially a successively refined maximum likelihood estimation based on the local graph structure.

II. CONSTRUCTION

This section describes the construction and corresponding encoding/decoding algorithms that lead to the result stated in Theorem I.1. A caricature of the data structure is portrayed in Figure 1.

![Fig. 1. An example of the data structure.](image)

**Overall Structure.** Let $L$ denote the number of layers. Let $V_0 = \{1, \ldots, N\}$, with $i \in V_0$ corresponding to $x_i$. Each index or node $i \in V_0$ has a counter of $b_0$ bits allocated to it. Because of this association, in what follows, we use the terms node (or vertex) and counter interchangeably, e.g., sometimes we call the $i^{th}$ node (or vertex) in $V_0$ the $i^{th}$ counter in $V_0$, and vice versa. Intuitively, the $b_0$ bits corresponding to $i^{th}$ node or index of $V_0$ are used to store the least significant $b_0$ bits of $x_i$. In order to store additional information, i.e., more significant bits of $x_i$'s collectively, layers 1 to $L$ are utilized. To this end, let $V_\ell$ denote a collection of indices or nodes corresponding to layer $\ell$, $1 \leq \ell \leq L$, with $|V_\ell| \leq |V_{\ell-1}|$ for $1 \leq \ell \leq L$. Each node $i \in V_\ell$ is allocated $b_\ell$ bits. The nodes of layers $\ell - 1$ and $\ell$, i.e., $V_{\ell-1}$ and $V_\ell$, are associated by a bipartite graph $G_\ell = (V_{\ell-1}, V_\ell, E_\ell)$ for $1 \leq \ell \leq L$. We denote the $i^{th}$ counter (stored at the $i^{th}$ node) in $V_\ell$ by $(i, \ell)$, and use $c(i, \ell)$ to denote the value stored by counter $(i, \ell)$.

**Setting Parameters.** Now we define parameters $L, b_\ell$ and $|V_\ell|$ for $0 \leq \ell \leq L$. Later we shall describe graphs $G_{\ell}, 1 \leq \ell \leq L$. Let $C$ and $K \geq 3$ be positive integers, and let $\varepsilon = \frac{1}{K}$. Parameters $C$ and $\varepsilon$ control the excess space used by the data structure compared to the information theoretic limit. Set $L$ as

$$L = \left( \log \log \left( \frac{B_2}{2K^2B_1^2} \right) - \log \log (B_1) \right)^+ + C.$$  

Also set $b_0 = \log(K^2B_1)$. Let $|V_1| = \varepsilon N$, and let $b_1 = \log(12KB)$. For $2 \leq \ell \leq L - 1$, $|V_\ell| = 2^{-(\ell-1)}\varepsilon N$, and $b_\ell = 3$. Finally, $|V_L| = 2^{-(L-1)}\varepsilon N$, and $b_L = \log \left( \frac{B_2}{2K^2B_1^2} \right)$.  

**Graph Description.** Here we describe the graphs $G_{\ell}, 1 \leq \ell \leq L$. Recall that we want these graphs to have a succinct description, efficient look up, large girth and ‘enough’ randomness. With these properties in mind, we start by describing the construction of $G_L$. We start with a bipartite graph $H$ with properties: (a) $H$ has $\sqrt{|V_{L-1}|}$ vertices on the left and $\sqrt{|V_{L-1}|}$ vertices on the right; and (b) $H$ is $(3,6)$-regular. The construction of such an $H$ will be described later, but first we use it to define $G_L$. Let $\tilde{G}$ be a graph consisting of $\sqrt{|V_{L-1}|}$ disjoint copies of $H$. Formally, $\tilde{G}$ has left vertices $v_0, v_1, \ldots, v_{|V_{L-1}|-1}$ and right vertices $w_0, w_1, \ldots, w_{|V_{L-1}|-1}$. Connect $v_0, v_1, \ldots, v_{|V_{L-1}|-1}$ to $w_0, w_1, \ldots, w_{|V_{L-1}|-1}$ using $H$. Then, connect $v_{|V_{L-1}|-1}^+, v_{|V_{L-1}|-1}^+, \ldots, v_{|V_{L-1}|-1}^+$ to $w_{|V_{L-1}|-1}+\sqrt{|V_{L-1}|}, \ldots, w_{|V_{L-1}|-1}+\sqrt{|V_{L-1}|}$ using $H$, and so on. $G_L$ is constructed to be isomorphic to $\tilde{G}$. Specifically, every left node $i \in \{0, \ldots, |V_{L-1}|-1\}$ of $G_L$ inherits the neighborhood of left node $Q(i) \in \{0, \ldots, |V_{L-1}|-1\}$ of $\tilde{G}$, where $Q(i)$ is defined as follows:

1. Let $M : \{0, \ldots, |V_{L-1}|-1\} \rightarrow \sqrt{|V_{L-1}|} \times \sqrt{|V_{L-1}|}$ be such that $M(i) = (r(i), c(i))$ with

$$r(i) = i \mod \sqrt{|V_{L-1}|} \quad \text{and} \quad c(i) = \left( i + \frac{i}{\sqrt{|V_{L-1}|}} \right) \mod \sqrt{|V_{L-1}|}.$$  

2. Let random map $R : \sqrt{|V_{L-1}|} \times \sqrt{|V_{L-1}|} \rightarrow \sqrt{|V_{L-1}|} \times \sqrt{|V_{L-1}|}$ be defined as $R(x, y) = (\pi_1(x), \pi_2(y))$ where $\pi_1, \pi_2$ are two permutations of length $\sqrt{|V_{L-1}|}$ chosen independently, uniformly at random.

3. Then, $Q = M^{-1} \circ R \circ M$.

The construction of $G_L, 2 \leq \ell \leq L - 1$ follows a very similar procedure with a different number of copies of $H$ used to construct the associated $\tilde{G}$ to account for the varying size of $V_\ell$ (and of course, new randomness to define the corresponding
We skip details due to space constraints. Finally, $G_1$ is constructed in essentially the same way as $G_L$, but with a different $(3,K)$-regular base graph $H_1$. The rest of the construction is identical. Finally, the base graphs $H$ and $H_1$ are based on Lubotzky et al.’s construction of Ramanujan graphs [16]. Again, due to space constraints we skip the details of the precise choice of parameters. We note that the earlier work by Vontobel et al. uses such graphs in the context of LDPC codes. However, we require an important modification of this construction to suit our needs.

In summary, thus constructed graphs $G_\ell, 1 \leq \ell \leq L$ have the following properties.

**Lemma II.1** For $1 \leq \ell \leq L$, $G_\ell$ has (a) girth $O(\log(N))$, (b) storage space $O(\sqrt{N} \log N)$, and (c) for any node $v$ (in the left or right partition), its neighbors can be computed in $O(1)$ time.

**An Example: Encoding/Decoding.** Here we explain how the described data structure can be used to store integers. Specifically, we will describe encoding and decoding algorithm. To this end, consider situation when $N = 4$ and $L = 2$. That is, $V_0 = \{1, 2, 3, 4\}$. Let $V_1, V_2$ be such that $|V_1| = 2$ and $|V_2| = 1$. Let $b_0 = 2$, $b_1 = 2$ and $b_2 = 2$ and initially all counters are set to 0. Now we explain the encoding algorithm.

We wish to write $x_3 = 2$, or equivalently we wish to increment $x_3$ by 2. For this, we add 2 to $c(3, 0)$. Since there are two bits in layer 0 in that position, its capacity is 4. Therefore, we do addition of 2 to the current value modulo 4. That is, the value in those two bits is updated to $0 + 2 \mod 4 = 2$. The resulting ‘overflow’ or ‘carry’ is $\lfloor (0 + 2)/4 \rfloor = 0$. Since it is 0, no further change is performed. This is shown in Figure 2(a).

Now, suppose $x_3$ is increased further by 3. Then, $c(3, 0)$ is changed to $2 + 3 \mod 4 = 1$ and ‘carry’ $\lfloor (2 + 3)/4 \rfloor = 1$ is added to the counters in layer 1 that are connected to $(3, 0)$ via graph $G_1$, i.e. counter $(1, 1)$. Repeating the same process, $c(1, 1)$ is changed to $0 + 1 \mod 4 = 1$, the resulting carry is 0 and hence no further change is performed. This is shown in Figure 2(b). Using a similar approach, writing $x_2 = 12$ will lead to Figure 2(c). Algorithm to decrement any of the $x_i$ is exactly the same as that for increment, but with negative value.

Finally, we give a simplified example illustrating the intuition behind the decoding algorithm. Figure 2(d) shows the initial configuration. The decoding algorithm uses the values stored in the counters to compute upper bounds and lower bounds on the counters. To see how this can be done for the simple example above, let us try to compute upper and lower bounds on $x_3$. First, observe that the counter $(1, 2)$ stores 0. Therefore, both counters in $V_1$ did not overflow. Thus, each counter in $V_1$ must already be storing the sum of the overflows of its neighbors in $V_0$.

We know $c(3, 0)$, so the problem of determining upper and lower bounds on $x_3$ is equivalent to determining upper and lower bounds on the overflow of counter $(3, 0)$. To determine a lower bound, consider the tree obtained by doing a breadth-first search of depth 2 in $G_1$ starting from $(3, 0)$, as shown in Figure 3(a). A trivial lower bound for the overflow of the bottom two counters is 0. Therefore, $1 - 0 - 0 = 1$ must be an upper bound on the overflow of $(3, 0)$.

Next, consider the graph obtained by doing a breadth-first search of depth 4 in $G_1$ starting from $(3, 0)$. This graph is not a tree, but we can pretend that it is a tree by making copies of nodes that are reached by more than one path, as shown in Figure 3(b). As before, 0 is a trivial lower bound for the overflow of the counters at the bottom of the tree. Therefore, $2 - 0 - 0 = 2$ is an upper bound for both counters at depth 2 in the tree. This means that $1 - 2 - 2 = -3$ must be a lower bound on the overflow of $(3, 0)$. Of course, in this case the trivial lower bound of 0 is better than the bound of $-3$ obtained by this reasoning. Since our upper bound on the overflow is 1.
and our lower bound is 0, we haven’t recovered the value of counter \((3, 0)\)’s overflow. However, in general, if this type of reasoning leads to matching upper and lower bounds, then the value of \((3, 0)\)’s overflow clearly must be equal to the value given by the matching upper and lower bounds. One can view our analysis later in the paper as showing that if we choose the graphs and counter sizes properly, then in fact it is extremely likely that the reasoning above does give matching upper and lower bounds, and thus we can use this kind of reasoning to construct an efficient decoding algorithm.

III. ENCODING AND DECODING ALGORITHMS

The encoding or update algorithm is the same as that of [21]. The example of Section II provides enough intuition to reconstruct the formal description of the encoding algorithm. Therefore, we skip details due to space constraints. In what follows, we provide a detailed description of the decoding algorithm that decodes each element w.h.p. It is based on the message passing algorithm proposed by [21]. However, to achieve the local decoding property, we carefully design an incremental version of the algorithm. Our analysis (unfortunately, skipped here due to space constraints) establishes this property. We call this algorithm WHP.

The idea behind WHP is that to decode a single input, we only need to use local information, e.g., instead of looking at all counters, we only use the values of counters that are close (in terms of graph distance) to the input we care about. As illustrated in Section II, use of local information provides upper and lower bounds for a given input. Following this insight, the algorithm can provide better upper and lower bounds if more and more local information (w.r.t. graphical code) is utilized. Therefore, the algorithm can incrementally use more information until it manages to decode (i.e., obtains matching upper and lower bounds). This is precisely the insight behind our algorithm.

Formally, the description of the WHP algorithm is stated as follows. First, a subroutine is stated that can recover the overflow of a single counter in an arbitrary layer of the data structure, given an additional information not directly stored in our structure – the values in the next layer assuming the next layer has finite sized counters. This subroutine is used by the ‘basic algorithm’ that utilizes more and more local information incrementally as discussed above. Now, the assumption utilized by the subroutine is not true for any layer \(\ell < L\), but holds true for the last layer \(L\) under our construction (i.e., the last layer never overflows as long as \(x \in S(B_1, B_2, N)\)). Therefore, using this subroutine, one can recursively obtain values in layers \(L - 1, \ldots, 0\). The ‘final algorithm’ achieves this by selecting appropriate parameters in the ‘basic algorithm’.

WHP: The Subroutine. For any counter \((u, \ell)\), the goal is to recover the value that it would have stored if \(b_\ell = \infty\). To this end, suppose the values of the counters in \(V_{\ell+1}\) are known with \(b_{\ell+1} = \infty\). As explained before, this will be satisfied recursively, starting with \(\ell + 1 = L\). The subroutine starts by computing the breadth first search (BFS) neighborhood of \((u, \ell)\) in \(G_{\ell+1}\) of depth \(t\), for an even \(t\). Hence the counters at depth \(t\) of this BFS or computation tree belong to \(V_t\) (root is at depth 0). This neighborhood is indeed a tree provided that \(G_{\ell+1}\) does not contain a cycle of length \(\leq 2t\). We will always select \(t\) so that this holds. This computation tree contains all the counters that are used by subroutine to determine the value that \((u, \ell)\) would have had if \(b_\ell = \infty\). Given this tree, the overflow of the root \((u, \ell)\) can be determined using the upper bound-lower bound method described by example in Section II. We skip detailed description of this method due to space constraints, but some intuition follows.

Imagine that the edges of the computation tree are oriented to point towards the root, so that it makes sense to talk about incoming and outgoing edges. Let \(C((u, \ell))\) denote the set of all children of counter \((u, \ell)\), and let \(P((u, \ell))\) denote the parent of counter \((u, \ell)\), with similar notation for counters in \(V_{\ell+1}\). Also, for a counter \((v, \ell + 1) \in V_{\ell+1}\), let \(c_v\) denote the value that this counter would have had if \(b_{\ell+1} = \infty\), which we recall was assumed to be available.

The algorithm passes a message (a number) \(m_{(u, \ell)} \rightarrow (w, \ell + 1)\) along the directed edges pointing from counter \((u, \ell)\) to counter \((w, \ell + 1)\) and similarly for edges pointing from a counter in \(V_{\ell+1}\) to a counter in \(V_t\) in this directed computation tree. The messages are started from the counters at depth \(t\), that belong to \(V_t\). These are equal to 0. Recursively, messages are propagated upwards in this computation tree as explained earlier in the example in Section II. Therefore, eventually the root will receive upper and lower bound values for trees of depth \(t\) and \(t + 2\).

WHP: The Basic Algorithm. Using repeated calls to the subroutine described above, the basic algorithm determines the value of \(x_i\).

The basic algorithm has a parameter for each layer, which we denote by \(t_\ell^{(1)}\). The algorithm proceeds as follows. Assume that we want to recover \(x_i\). Then, we start by building the computation tree of depth \(t_1^{(1)} + 2\) in the first layer, i.e., computation tree formed by a breadth-first search of the graph \(G_0\) rooted at \((i, 0)\). Next, for each counter in this computation tree that also belongs to \(V_1\), i.e., the counters at odd depth in the computation tree, we form a new computation tree of depth \(t_2^{(1)} + 2\) in the second layer. This gives us a set of computation trees in the second layer. For each computation tree, the counters at odd depth belong to \(V_3\), so we repeat the process and build a computation tree of depth \(t_3^{(1)} + 2\) in the third layer for each of these counters. We repeat this process until we reach the last layer, layer \(L\). Figure 4 gives an example of this process.

Now, we work backwards from layer \(L\). Each computation tree in layer \(L\) has depth \(t_L^{(1)} + 2\). We run the subroutine described above on each of these computation trees. When we run the subroutine on a particular computation tree, we set the input \(c_v\) for each counter \((v, L)\) in \(V_L\) that also belongs to the computation tree to be \(c((v, L))\). Intuitively, this means that we are assuming that no counter in \(V_L\) overflows. As mentioned
earlier, under our construction for any $x \in S(B_1, B_2, N)$, this holds true. Details are skipped due to space constraints.

Assuming that none of the subroutine calls fail, we have computed the overflow for all the counters that appear in computation trees for layer $L-1$. Let $(v, L-1)$ be a counter in $V_{L-1}$ that also belongs to a computation tree in layer $L-1$. We have computed $(v, L-1)$’s overflow, which we denote by overflow$(v, L-1)$. To compute the value that $(v, L-1)$ would have stored if $b_{L-1} = \infty$, we simply compute $c_v = c(v, L-1) + 2^{b_v-1}$ overflow$(v, L-1)$. Once we have computed these values, we can run the subroutine on the computation trees for layer $L-1$. Then, we compute the appropriate values for counters in $V_{L-2}$ using the formula

$$c_v = c(v, L-2) + 2^{b_v-2}$overflow(v, L-2),$$

and so on until either the subroutine fails or we successfully run the subroutine on all the computation trees. Assuming that the subroutine finishes successfully on all of the computation trees, the final subroutine call gives us the overflow of $(i, 0)$. Then,

$$x_i = c(i, 0) + 2^{b_0}$overflow(i, 0).$$

Thus, if none of the subroutine calls fail, then we successfully recover $x_i$.

**WHP: The Final Algorithm.** The final algorithm repeats the basic algorithm several times. Specifically, the final algorithm starts by running the basic algorithm. If none of the subroutine calls fail, then we recover $x_i$ and we are done. Otherwise, we run the basic algorithm again, but with a new set of parameters $t_1^{(p)}, t_2^{(p)}, \ldots, t_L^{(p)}$. If some subroutine call fails, we run the basic algorithm again with a new set of parameters $t_1^{(3)}, t_2^{(3)}, \ldots, t_L^{(3)}$, and so on up to some maximum number $M$ of repetitions. If after $M$ repetitions we still fail to recover $x_i$, then we declare failure. For our purposes, we set the parameters as follows. To specify $t_{\ell}^{(p)}$, we introduce the auxiliary numbers $\delta_{\ell}^{(p)}$ and $n_{\ell}^{(p)}$. For $1 \leq p \leq \frac{\log(L)}{\log(\log(L))}$, $1 \leq \ell \leq L$, we define $\delta_{\ell}^{(p)}$, $n_{\ell}^{(p)}$, and $t_{\ell}^{(p)}$ as follows:

$$\delta_{\ell}^{(p)} = e^{-L^{p+1}}$$

$$n_{\ell}^{(p)} = e^{f\left(c_1 + \frac{\log(c)}{\log(\log(L))} \log\left(\frac{1}{\delta_{\ell}^{(p)}}\right)\right)}$$

$$t_{\ell}^{(p)} = \left[t^* + \frac{1}{\log d} \log\left(\frac{1}{a} \log\left(\frac{n_{\ell}^{(p)} L}{\delta_{\ell}^{(p)}}\right)\right)\right],$$

where $a, c, c_1, d$, and $t^*$ are some fixed constants. Finally, we set $M = \log(L) / \log(\log(L))$. This completes the specification of all the parameters.

**References**


[38] Michael L. Fredman, Janos Komlos, and Endre Szemeredi. Storing a sparse table with 0(1) worst case access time. Journal of the ACM, 31(3):538-544, 1984. See also FOCS82.

