Compressed Dynamic Range Majority Data Structures

Travis Gagie[1,2], Meng He[3], and Gonzalo Navarro[2,4]

[1] Diego Portales University   [2] CeBiB   [3] Dalhousie University   [4] DCC, University of Chile
travis.gagie@mail.udp.cl   mhe@cs.dal.ca   gnavarro@dcc.uchile.cl

Abstract

In the range α-majority query problem, we preprocess a given sequence $S[1..n]$ for a fixed threshold $\alpha \in (0, 1]$, such that given a query range $[i..j]$, the symbols that occur more than $\alpha(j-i+1)$ times in $S[i..j]$ can be reported efficiently. We design the first compressed solution to this problem in dynamic settings. Our data structure represents S using $nH_k + o(n \lg \sigma)$ bits for any $k = o(\log_\sigma n)$, where σ is the alphabet size and H_k is the k-th order empirical entropy of S. It answers range α-majority queries in $O(\frac{\lg n}{\alpha \lg \lg n})$ time, and supports insertions and deletions in $O(\frac{\lg n}{\alpha})$ amortized time. The best previous solution [1] has the same query and update times, but uses $O(n)$ words.

1   Introduction

Given a threshold $\alpha \in (0, 1]$, a symbol c is an α-*majority* in a sequence $S[1..n]$ if c occurs more than αn times in S. Thus α-majorities are often used to represent frequent symbols and, naturally, the problem of finding α-majorities is important in data mining [2, 3]. Misra and Gries [4] proposed an optimal solution that computes all α-majorities using $O(n \lg(1/\alpha))$ comparisons, and when implemented in word RAM for a sequence over an alphabet of size σ, the running time becomes $O(n)$ [3].

In the *range α-majority query* problem, we further preprocess S such that given a query range $[i..j]$, the α-majorities of $S[i..j]$, i.e., the symbols that occur more than $\alpha(j-i+1)$ times in $S[i..j]$, can be reported efficiently. Karpinski and Nekrich [5] first considered this problem and proposed a solution that uses $O(n/\alpha)$ words to support queries in $O((\lg \lg n)^2/\alpha)$ time. Durocher *et al.* [6] presented the first solution that achieves $O(1/\alpha)$ optimal query time, and their structure occupies $O(n/\alpha)$ words. Subsequently, much work has been done to make the space cost independent of α [7, 8, 9], and even to achieve compression [7, 9] when S is drawn from a fixed alphabet. For example, Gagie *et al.* [9] showed how to represent S using $(1 + \epsilon)nH_0 + o(n)$ bits for any constant $\sigma \in (0, 1)$ to answer range α-majority queries in $O(1/\alpha)$ time, where H_0 is the 0-th order empirical entropy of S. We refer to [9] for a thorough survey.

In dynamic settings, we wish to maintain S to support range α-majority queries under the following update operations: i) `insert`(c, i), which inserts symbol c between $A[i-1]$ and $A[i]$, shifting the symbols in positions i to n to positions $i+1$ and $n+1$, respectively; ii) `delete`(c, i), which deletes $A[i]$, shifting the symbols in positions i to n to positions $i-1$ and $n-1$, respectively. Elmasry *et al.* [1] considered this problem,

---

and they designed an $O(n)$-word structure that can answer range α-majority queries in $O(\frac{\lg n}{\alpha \lg \lg n})$ time, and supports insertions and deletions in $O(\frac{\lg n}{\alpha})$ amortized time.[1]

Previously, no succinct data structures have been designed for dynamic range α-majorities. We thus design the first compressed data structure for this problem. Our data structure represents S using $nH_k + o(n \lg \sigma)$ bits for any $k = o(\log_\sigma n)$, where σ is the alphabet size and H_k is the k-th order empirical entropy of S. It answers range α-majority queries in $O(\frac{\lg n}{\alpha \lg \lg n})$ time, and supports insertions and deletions in $O(\frac{\lg n}{\alpha})$ amortized time. Hence, its query and update times match the best previous solution by Elmasry *et al.* [1], while using compressed space.

2   Preliminaries

We first summarize some existing data structures that will be used in our solution. One such data structure is designed for the problem of maintaining a string S under `insert` and `delete` operations to support the following operations: `access`(i), which returns $S[i]$; `rank`(c, i), which returns the number of occurrences of character c in $S[1..i]$; and `select`(c, i), which returns the position of the i-th occurrence of c in S. The following lemma summarizes the currently best compressed solution to this problem, which also supports the extraction of an arbitrary substring in optimal time:

Lemma 1 ([10]). *A string of length n over an alphabet of size σ can be represented using $nH_k + o(n \lg \sigma)$ bits for any $k = o(\log_\sigma n)$ to support* `access`, `rank`, `select`, `insert` *and* `delete` *in $O(\lg n / \lg \lg n)$ time. It also supports the extraction of a substring of length l in $O(\lg n / \lg \lg n + l / \log_\sigma n)$ time.*

Raman *et al.* [11] considered the problem of representing a dynamic integer sequence Q to support the following operations: `sum`(Q, i), which computes $\sum_{j=1}^{i} Q[j]$; `search`(Q, x), which returns the smallest i with `sum`$(Q, i) \geq x$; and `update`(Q, i, δ), which sets $Q[i]$ to $Q[i] + \delta$. One building component of their solution is a data structure for small sequences, which will also be used in our data structures:

Lemma 2 ([11]). *A sequence, Q, of $O(\lg^\epsilon n)$ nonnegative integers of $O(\lg n)$ bits each, where $0 \leq \epsilon < 1$, can be represented using $O(\lg^{1+\epsilon} n)$ bits to support* `sum`, `search`, *and* `update`(Q, i, δ) *where $|\delta| \leq \lg n$, in $O(1)$ time. This data structure can be constructed in $O(\lg^\epsilon n)$ time, and requires a precomputed universal table occupying $O(n^{\epsilon'})$ bits for any fixed $\epsilon' > 0$.*

3   Compressed Dynamic Range Majority Data Structures

In this section we design compressed dynamic data structures for range α-majority queries. We define three different types of queries as follows. Given an α-majority

---

[1]Karpinski and Nekrich [5] also considered the dynamic case, though they defined the data set as a set of colored points in 1D. With a reduction developed in [1], the solutions by Karpinski and Nekrich can also be used to encode dynamic sequences, though the results are inferior to those of Elmasry *et al.* [1].

query with range $[i..j]$, we compute the size, r, of the query range as $j - i + 1$. If $r \geq L$, where $L = \lceil \frac{1}{\alpha}(\lceil \frac{\lg n}{\lg \lg n} \rceil)^2 \rceil$, then we say that this query is a *large-sized query*. The query is called a *medium-sized query* if $L' < r < L$, where $L' = \lceil \frac{1}{\alpha} \lceil \frac{\lg n}{\lg \lg n} \rceil \rceil$. If $r \leq L'$, then it is a *small-sized query*.

We represent the input sequence S using Lemma 1. This supports small-sized queries immediately: By Lemma 1, we can compute the content of the subsequence $S[i..j]$, where $[i..j]$ is the query range, in $O(\frac{\lg n}{\lg \lg n} + \frac{j-i+1}{\log_\sigma n}) = O(\frac{\lg n}{\alpha \lg \lg n})$ time. We can then compute the α-majorities in $S[i..j]$ in $O(j - i + 1) = O(\frac{\lg n}{\alpha \lg \lg n})$ time using the algorithm of Misra and Gries [4]. Thus it suffices to construct additional data structures for large-sized and medium-sized queries.

3.1 Supporting Large-Sized Range α-Majority Queries

To support large-sized queries, we construct a weight-balanced B-tree [12] T with branching parameter 8 and leaf parameter L. We augment T by adding, for each node, a pointer to the node immediately to its left at the same level, and another pointer to the node immediately to its right. These pointers can be maintained easily under updates, and will not affect the space cost of T asymptotically. Each leaf of T represents a contiguous subsequence, or *block*, of S, and the entire sequence S can be obtained by concatenating all the blocks represented by the leaves of T from left to right. Each internal node of T then represents a block that is the concatenation of all the blocks represented by its leaf descendants. We number the levels of T by $0, 1, 2, \ldots$ from the leaf level to the root level. Thus level a is higher than level b if $a > b$. Let v be a node at the l-th level of T, and let $B(v)$ denote the block it represents. Then, by the properties of weight-balanced B-trees, if v is a leaf, the length of its block, denoted by $|B(v)|$, is at least L and at most $2L - 1$. If v is an internal node, then $\frac{1}{2} \cdot 8^l \cdot L < |B(v)| < 2 \cdot 8^l \cdot L$. We also have that each internal node has at least 2 and at most 32 children.

We do not store the actual content of a block in the corresponding node of T. Instead, for each v, we store the size of the block that it represents, and in addition, compute and store information in a structure $C(v)$ called *candidate list* about symbols that can possibly be the α-majorities of subsequences that meet certain conditions. More precisely, let l be the level of v, u be the parent of v, and $SB(v)$ be the concatenation of the blocks represented by the node immediately to the left of u at level $l + 1$, the node u, and the node immediately to the right of u at level $l + 1$. Then $C(v)$ contains each symbol that appears more than αb_l times in $SB(v)$, where $b_l = \frac{1}{2} \cdot 8^l \cdot L$ is the minimum size of a block at level l. Since the maximum length of each block at level $l + 1$ is $4b_{l+1} = 32b_l$, we have $|SB(v)| \leq 96b_l$, and thus $|C(v)| = O(1/\alpha)$. To show the idea behind the candidate lists, we say that two subsequences *touch* each other if their corresponding sets of indices in S are not disjoint. We then observe that, since the size of any block at level $l + 1$ is greater than $8b_l$, any subsequence $S[i..j]$ touching $B(v)$ is completely contained in $SB(v)$ if $r = j - i + 1$ is within $(b_l, 8b_l)$. Since each α-majority in $S[i..j]$ appears at least $\alpha r > \alpha b_l$ times, it is also contained in $C(v)$. Therefore, to find the α-majority in $S[i..j]$, it suffices to verify

whether each element in $C(v)$ is indeed an answer; more details are to be given in our query algorithm later.

Even though it only requires $O(|SB(v)|)$ time to construct $C(v)$ [4], it would be costly to reconstruct it every time an update operation is performed on $SB(v)$. To make the cost of maintaining $C(v)$ acceptable, we only rebuild it periodically by adopting a strategy by Karpinski and Nekrich [5]. More precisely, when we construct $C(v)$, we store symbols that occur more than $\alpha b_l/2$ times in $SB(v)$. We also keep a counter $U(v)$ that we increment whenever we perform insert or delete in $SB(v)$. Only when $U(v)$ reaches $\alpha b_l/2$ do we reconstruct C_B, and then we reset $U(v)$ to 0. Since at most $\alpha b_l/2$ updates can be performed to $|SB(v)|$ between two consecutive reconstructions, any symbol that becomes an α-majority in $|SB(v)|$ any time during these updates must have at least $\alpha b_l/2$ occurrences in $SB(v)$ before these updates are performed. Thus we can guarantee that any symbol that appears more than αb_l times in $SB(v)$ is always contained in $C(v)$ during updates. The size of $C(v)$ is still $O(b_l/\alpha)$, and, as to be shown later, it only requires $O((\lg n)/\alpha)$ amortized time per update to S to maintain all the candidate lists.

We also construct data structures to speed up a top-down traversal in T. These data structures are defined for the *marked* levels of T, where the k-th marked level is level $k\lceil(1/6)\lg\lg n\rceil$ of T for $k = 0, 1, \ldots$. Given a node v at the k-th marked level, the number of its descendants at the $(k-1)$-st marked level is at most $32^{\lceil(1/6)\lg\lg n\rceil-1} \leq 32^{(1/6)\lg\lg n} = \lg^{5/6} n$. Thus, the sizes of the blocks represented by these descendants, when listed from left to right, form an integer sequence, $Q(v)$, of at most $\lg^{5/6} n$ entries. We represent $Q(v)$ using Lemma 2, and store a sequence of pointers $P(v)$, in which $P(v)[i]$ points to the i-th leftmost descendant at the $(k-1)$-st marked level.

We next prove the following key lemma regarding an arbitrary subsequence $S[i..j]$ of length greater than L, which will be used in our query algorithm:

Lemma 3. *If $r = j - i + 1 > L$, then each α-majority element in $S[i..j]$ is contained in $C(v)$ for any node v at level $l = \lceil\frac{1}{3}\lg\frac{2r}{L} - 1\rceil$ whose block touches $S[i..j]$.*

Proof. Let u be v's parent. Then $S[i..j]$ also touches u, and u is at level $l+1$. Let u_1 and u_2 be the nodes immediately to the left and right of u at level $l+1$, respectively.

Let b_l and b_{l+1} denote the minimum block size represented by nodes at level l and $l+1$ of T, respectively. Then, by the properties of weight-balanced B-trees, if $l > 0$, $b_l = \frac{1}{2}\cdot 8^l\cdot L = \frac{1}{2}\cdot 8^{\lceil\frac{1}{3}\lg\frac{2r}{L}-1\rceil}\cdot L < \frac{1}{2}\cdot 8^{\frac{1}{3}\lg\frac{2r}{L}}\cdot L = r$. When $l = 0$, $b_l = L < r$. Thus, we always have $b_l < r$. Therefore, any α-majority of $S[i..j]$ occurs more than $\alpha r > \alpha b_l$ times in $S[i..j]$.

On the other hand, $b_{l+1} = \frac{1}{2}\cdot 8^{\lceil\frac{1}{3}\lg\frac{2r}{L}\rceil}\cdot L \geq \frac{1}{2}\cdot 8^{\frac{1}{3}\lg\frac{2r}{L}}\cdot L = r$. Since $S[i..j]$ touches $B(u)$, this inequality means that $S[i..j]$ is entirely contained in either the concatenation of $B(u_1)$ and $B(u)$, or the concatenation of $B(u)$ and $B(u_2)$. In either case, $S[i..j]$ is contained in $SB(v)$. Since any α-majority of $S[i..j]$ occurs more than αb_l times in $S[i..j]$, it also occurs more than αb_l times in $SB(v)$. As $C(v)$ includes any symbol that appears more than αb_l times in $SB(v)$, any α-majority of $S[i..j]$ is contained in $C(v)$. $\qquad\square$

We now describe our query and update algorithms, and analyze space cost.

Lemma 4. *Large-sized range α-majority queries can be supported in $O(\frac{\lg n}{\alpha \lg \lg n})$ time.*

Proof. Let $[i..j]$ be the query range, $r = j - i + 1$ and $l = \lceil \frac{1}{3} \lg \frac{2r}{L} - 1 \rceil$. We first look for a node v at level l whose block touches $S[i..j]$. The obvious approach is to perform a top-down traversal of T to look for a node at level l whose block contains position i. During the traversal, we make use of the information about the lengths of the blocks represented by the nodes of T to decide which node at the next level to descend to, and to keep track of the starting position in S of the block represented by the node that is currently being visited. More precisely, suppose that we visit node u at the current level as we have determined previously that $B(u)$ contains $S[i]$. We also know that the first element in $B(u)$ is $S[p]$. Let $u_1, u_2, \ldots, u_d$ denote the children of u, where $d \leq 32$. To decide which child of u represents a block that contains $S[i]$, we retrieve the lengths of all $|B(u_k)|$'s, and look for the smallest q such that $\sum_{k=1}^{q} |B(u_k)| \geq i$. Node u_q is then the node at the level below whose block contains $S[i]$, and the starting position of its block in S is $p + \sum_{k=1}^{q-1} |B(u_k)|$. As $d \leq 32$ and we store the length of the block that each node represents, these steps use $O(1)$ time.

If we follow the approach described in the previous paragraph, we will use $O(\lg n)$ time in total, as T has $O(\lg n)$ levels. Thus we make use of the additional data structures stored at marked levels to speed up this process. If there is no marked level between the root level and l, then the top down traversal only descends $O(\lg \lg n)$ levels, requiring $O(\lg \lg n)$ time only. Otherwise, we perform the top-down traversal until we reach the highest marked level. Let x be the node that we visit at the highest marked level. As $Q(x)$ stores the lengths of the blocks at the next marked level, we can perform a `search` operation in $Q(x)$ and then follow an appropriate pointer in $P(x)$ to look for the node y at the second highest level that contains $S[i]$, and perform a `sum` operation in $Q(x)$ to determine the starting position of $B(y)$ in S. These operations require $O(1)$ time. We repeat this process until we reach the lowest marked level above level l, and then we descend level by level until we find node v. As there are $O(\lg n / \lg \lg n)$ marked levels, the whole process takes $O(\lg n / \lg \lg n)$ time.

By Lemma 3, we know that the α-majorities of $S[i..j]$ are contained in $C(v)$. We then verify, for each symbol, c, in $C(v)$, whether it is indeed an α-majority by computing its number, m, of occurrences in $S[i..j]$ and comparing m to αr. As $m = \texttt{rank}(c, j) - \texttt{rank}(c, i - 1)$, m can be computed in $O(\lg n / \lg \lg n)$ time by Lemma 1. As $|C(v)| = O(1/\alpha)$, it requires $O(\frac{\lg n}{\alpha \lg \lg n})$ time in total to find out which of these symbols should be included in the answer to the query. Therefore, the total query time is $O(\frac{\lg n}{\lg \lg n} + \frac{\lg n}{\alpha \lg \lg n}) = O(\frac{\lg n}{\alpha \lg \lg n})$. □

Lemma 5. *The data structures described in Section 3.1 can be maintained in $O(\frac{\lg n}{\alpha})$ amortized time under update operations.*

Proof. We only show how to support `insert`; the support for `delete` is similar.

To perform `insert`(c, i), we first perform a top down traversal to look for the node v at level 0 whose block contains $S[i]$. During this traversal, we descend level by level as in Lemma 4, but we do not use the marked levels to speed up the process. For each node u that we visit, we increment the recorded length of $B(u)$. In addition, we update the counters U stored in the children of u and in the children of the two

nodes that surround u. There are a constant number of these nodes, and they can all be located in $O(1)$ time by following either the edges of T, or the pointers between two nodes that are next to each other at the same level where we augment T.

When incrementing the counter U of each node, we find out whether the candidate list of this node has to be rebuilt. To reconstruct the candidate list of a node x at level l, we first compute the starting and ending positions of $SB(x)$ in S. This can be computed in $O(1)$ time because, during the top down traversal, we have already computed the starting and ending positions of $B(v)$ in S, and the three nodes whose blocks form $SB(x)$, as well as the sizes of these three blocks, can be retrieved by following a constant number of pointers starting from v. We then extract the content of $SB(x)$. As $|SB(x)| \leq 96b_l$ (see discussions earlier in this section) and $b_l \geq L$, by Lemma 1, $SB(x)$ can be extracted from S in $O(b_l)$ time. We next compute all the symbols that appear in $SB(x)$ more than $\alpha b_l/2$ times in $O(b_l)$ time [4], and these are the elements in the reconstructed $C(x)$. Since the counter $U(x)$ has to reach $\alpha b_l/2$ before $C(x)$ has to be rebuilt, the amortized cost per update is $O(1/\alpha)$.

If u is at a marked level, we perform a `search` operation in $O(1)$ time to locate the entry of $Q(u)$ that corresponds to the node at the next lower marked level whose block contains i, and perform an `update`, again in $O(1)$ time, to increment the value stored in this entry. So far we have used $O(1/\alpha)$ amortized time for each node we visit during the top-down traversal. Since T has $O(\lg n)$ levels, the overall cost we have calculated up to this point is $O((\lg n)/\alpha)$ amortized time.

When a node, z, at level l of T splits, we reconstruct $C(z)$ in $O(b_l)$ time. If l is a marked level, but it is not the lowest marked level, we also rebuild $Q(z)$ and $P(z)$ in $O(\lg^{1/6} n) = o(b_l)$ time. By the properties of a weight-balanced B-tree, after a node at level l has been split, it requires at least $\frac{1}{2} \cdot 8^l \cdot L = b_l$ insertions before it can be split again. Therefore, we can amortize the cost of reconstructing these data structures over the insertions between reconstructions, and each `insert` is thus charged with $O(1)$ amortized cost. As each `insert` may cause one node at each level of T to split, the overall cost charged to an `insert` operation is thus $O(\lg n)$.

Finally, update operations may cause the value of L to change, but then the value of $\lceil \frac{\lg n}{\lg \lg n} \rceil$ must change, after $\Omega(n)$ updates. It is clear that our data structures can be constructed in $O(n \lg n)$ time, incurring $O(\lg n)$ amortized time for each update. To summarize, `insert` can be supported in $O((\lg n)/\alpha)$ amortized time. $\qquad \square$

Lemma 6. *The data structures described in Section 3.1 occupy $o(n \lg \sigma)$ bits.*

Proof. As T has $O(n/L)$ nodes, the structure of T, pointers between nodes at the same level, as well as counters and block lengths stored with the nodes, occupy $O(n/L \times \lg n) = O(\frac{\alpha n (\lg \lg n)^2}{\lg n})$ bits in total. Each candidate list can be stored in $O((\lg \sigma)/\alpha)$ bits, so the candidate lists stored in all the nodes use $O(n/L \times (\lg \sigma)/\alpha) = O(\frac{n \lg \sigma (\lg \lg n)^2}{\lg^2 n})$ bits in total. The size of the structures $Q(v)$ and $P(v)$ can be charged to the pointed nodes, so there are $O(n/L)$ entries to store. As each entry of $Q(v)$ uses $O(\lg n)$ bits, all the $Q(v)$'s occupy $O(n/L \times \lg n) = O(\frac{\alpha n (\lg \lg n)^2}{\lg n})$ bits. The same analysis applies to $P(v)$. Therefore, the data structures described in this section use $O(\frac{\alpha n (\lg \lg n)^2}{\lg n} + \frac{n \lg \sigma (\lg \lg n)^2}{\lg^2 n}) = o(n \lg \sigma)$ bits. $\qquad \square$

3.2 Supporting Medium-Sized Range α-Majority Queries

We could use the same structures designed in Section 3.1 to support medium-sized queries if we simply set the leaf parameter of T to be L' instead of L, but then the resulting data structures would not be succinct. To save space, we build a data structure $D(v)$ for each leaf node v of T. Our idea for supporting medium-sized queries is similar to that for large-sized queries, but since the block represented by a leaf node of T is small, we are able to simplify the idea and the data structures. Such simplifications allow us to maintain a multi-level decomposition of $B(v)$ in a hierarchy of lists instead of in a tree, which are further laid out in one contiguous chunk of memory for each leaf node of T, to avoid using too much space for pointers.

We now describe this multi-level decomposition of $B(v)$, which will be used to define the data structure components of $D(v)$. As we define one set of data structure components in $D(v)$ for each level of this decomposition, we use $D(v)$ to refer to both the data structure that we build for $B(v)$ and the decomposition of $B(v)$. To distinguish a level of $D(v)$ from a level of T, we number each level of $D(v)$ using a non-positive integer. At level $-l$, for $l = 0, 1, 2, \ldots, \lceil \lg(L/L') - 1 \rceil$, $B(v)$ is partitioned into *mini-blocks* of length between $L/2^l$ and $L/2^{l-1}$. Note that the level 0 decomposition contains simply one mini-block, which is $B(v)$ itself, as the length of any leaf block in T is between L and $2L$ already. We define $m_l = L/2^l$, which is the minimum length of a mini-block at level $-l$. As $L' < m_{\lceil \lg(L/L') - 1 \rceil} \leq 2L'$, the minimum length of a mini-block at the lowest level, i.e., level $-\lceil \lg(L/L') \rceil - 1$, is between L' and $2L'$.

For each mini-block M at level $-l$ of $D(v)$, we define its *predecessor*, $\mathtt{pred}(M)$, as follows: If M is not the leftmost mini-block at level $-l$ of $D(v)$, then $\mathtt{pred}(M)$ is the mini-block immediately to its left at the same level. Otherwise, if v is not the leftmost leaf ($\mathtt{pred}(M)$ is null otherwise), let v_1 be the leaf immediately to the left of v in T, and $\mathtt{pred}(M)$ is defined to be the rightmost mini-block at level $-l$ of $D(v_1)$. Similarly, we define the *successor*, $\mathtt{succ}(M)$, of M as the mini-block immediately to the right of M at level $-l$ of $D(v)$ if such a mini-block exists. Otherwise, $\mathtt{succ}(M)$ is the leftmost mini-block at level $-l$ of $D(v_2)$ where v_2 is the leaf immediately to the right of v in T if v_2 exists, or null otherwise. Then, the candidate list, $C(M)$, of M contains each symbol that occurs more than $\alpha m_l/2$ times in the concatenation of M, $\mathtt{pred}(M)$ and $\mathtt{succ}(M)$. To maintain $C(M)$ during updates, we use the same strategy in Section 3.1 that is used to maintain $C(v)$. More specifically, we store a counter $U(M)$ so that we can rebuild $C(M)$ after exactly $\alpha m_l/4$ update operations have been performed to M, $\mathtt{pred}(M)$ and $\mathtt{succ}(M)$. Whenever we perform the reconstruction, we include in $C(M)$ each symbol that occurs more than $\alpha m_l/4$ times in the concatenation of M, $\mathtt{pred}(M)$ and $\mathtt{succ}(M)$. Since $|\mathtt{pred}(M)| + |M| + |\mathtt{succ}(M)| \leq 6m_l$, the number of symbols included in $C(M)$ is at most $24/\alpha$.

The precomputed information for each mini-block M includes $|M|$, $C(M)$, and $U(M)$. These data for mini-blocks at the same level, $-l$, of $D(v)$ are chained together in a doubly linked list $L_l(v)$. $D(v)$ then contains these $O(\lg(L/L')) = O(\lg \lg n)$ lists. We however cannot afford storing each list in the standard way using pointers of $O(\lg n)$ bits each, as this would use too much space. Instead, we lay them out in a contiguous chunk of memory as follows. We first observe that the number of mini-

blocks at level $-l$ of $D(v)$ is less than $2L/(L/2^l) = 2^{l+1}$. Thus, the total number of mini-blocks across all levels is less than $2 \cdot 2^{\lceil \lg(L/L')-1 \rceil + 1} - 1 < 4L/L'$. We then use an array $A(v)$ of $\lceil 4L/L' \rceil$ fix-sized *slots* to store $D(v)$, and each slot stores the precomputed information of a mini-block.

To determine the size of a slot, we compute the maximum number of bits needed to encode the precomputed information for each mini-block M. $C(M)$ can be stored in $\lceil \lg \sigma \rceil \cdot \lceil 24/\alpha \rceil$ bits. As M has less than $2L$ elements, its length can be encoded in $\lceil \lg(2L) \rceil$ bits. The counter $U(M)$ can be encoded in $\lceil \lg(\alpha m_l/4) \rceil < \lceil \lg(\alpha L/2) \rceil \le \lceil \lg(L/2) \rceil$ bits. The two pointers to the neighbours of M in the linked list can be encoded as the indices of these mini-blocks in the memory chunk. Since there are $\lceil 4L/L' \rceil$ slots, each pointer can be encoded in $\lceil \lg \lceil 4L/L' \rceil \rceil$ bits. Therefore, we set the size of each slot to be $\lceil \lg \sigma \rceil \cdot \lceil 24/\alpha \rceil + 2\lceil \lg L \rceil + 2\lceil \lg \lceil 4L/L' \rceil \rceil$ bits.

We prepend this memory chunk with a header. This header encodes the indices of the slots that store the head of each $L_l(v)$. As there are $\lceil \lg(L/L') \rceil$ levels and each index can be encoded in $\lceil \lg \lceil 4L/L' \rceil \rceil$ bits, the header uses $\lceil \lg(L/L') \rceil \cdot \lceil \lg \lceil 4L/L' \rceil \rceil$ bits. Clearly our memory management scheme allows us to traverse each doubly linked list $L_l(v)$ easily. When mini-blocks merge or split during updates, we need to perform insertions and deletions in the doubly linked lists. To facilitate these updates, we always store the precomputed information for all mini-blocks in $D(v)$ in a prefix of $A(v)$, and keep track of the number of used slots of $A(v)$. When we perform an insertion into a list $L_l(v)$, we use the first unused slot of A to store the new information, and update the header if the newly inserted list element becomes the head. When we perform a deletion, we copy the content of the last used slot (let M' be the mini-block that corresponds to it) into the slot corresponding to the deleted element of $L_l(v)$. We also follow the pointers encoded in the slot for M' to locate the neighbours of M' in its doubly linked list, and update pointers in these neighbours that point to M'. If M' is the head of a doubly linked list (we can determine which list it is using $|M'|$), we update the header as well. The following lemma shows that our memory management strategy indeed saves space:

Lemma 7. *The data structures described in Section 3.2 occupy $o(n \lg \sigma)$ bits.*

Proof. We first analyze the size of the memory chunk storing $D(v)$ for each leaf v of T. By our analysis in previous paragraphs, we observe that the header of this chunk uses $O((\lg \lg n)^2)$ bits. Each slot of $A(v)$ uses $O(\lg \sigma/\alpha + \lg \lg n)$ bits, and $A(v)$ has $O(\lg n / \lg \lg n)$ entries. Therefore, $A(v)$ occupies $O(\frac{\lg \sigma \lg n}{\alpha \lg \lg n} + \lg n)$ bits. Hence the total size of the memory chunk of each leaf of T is $O(\frac{\lg \sigma \lg n}{\alpha \lg \lg n} + \lg n)$ bits. As there are $O(n/L)$ leaves in T, the data structures described in this section uses $O(\frac{n \lg \sigma \lg \lg n}{\lg n} + \frac{\alpha n (\lg \lg n)^2}{\lg n}) = o(n \lg \sigma)$ bits. □

We now show how to support query and update operations.

Lemma 8. *Medium-sized range α-majority queries can be supported in $O(\frac{\lg n}{\alpha \lg \lg n})$ time.*

Proof. Let $[i..j]$ be the query range and let $r = j - i + 1$. We first perform a top down traversal in T to locate the leaf, v, that represents a block containing $S[i]$ in $O(\frac{\lg n}{\lg \lg n})$ time using the approach described in the proof of Lemma 4. In this process, we can also find the starting position of $B(v)$ in S.

We next make use of $D(v)$ to answer the query as follows. Let $l = \lceil \lg(L/r) - 1 \rceil$. As $m_l = L/2^{\lceil \lg(L/r) - 1 \rceil}$, we have $m_l/2 \leq r < m_l$. We then scan the list $L_l(v)$ to look for a mini-block, M, that contains $S[i]$ at level $-l$. This can be done by first locating the head of $L_l(v)$ from the header of the memory chunk that stores $D(v)$, and then perform a linear scan, computing the starting position of each mini-block in $L_l(v)$ along the way. As $L_l(v)$ has at most $O(L/L') = O(\frac{\lg n}{\lg \lg n})$ entries, we can locate M in $O(\frac{\lg n}{\lg \lg n})$ time. Since $m_l > r$, $S[i..j]$ is either entirely contained in the concatenation of $\texttt{pred}(M)$ and M, or the concatenation of M and $\texttt{succ}(M)$. Thus each α-majority of $S[i..j]$ must occur more than $\alpha r > \alpha m_l/2$ times in the concatenation of $\texttt{pred}(M)$, M and $\texttt{succ}(M)$. Therefore, each α-majority of $S[i..j]$ is contained in $C(M)$. We can then perform $\texttt{rank}$ operations in S to verify whether each symbol in $C(M)$ is indeed an α-majority of $S[i..j]$. As $C(M)$ has $O(1/\alpha)$ symbols, this process requires $O(\frac{\lg n}{\alpha \lg \lg n})$ time. $\qquad\square$

Lemma 9. *The data structures described in Section 3.2 can be maintained in $O(\frac{\lg n}{\lg \lg n} + \frac{\lg \lg n}{\alpha})$ amortized time under update operations.*

Proof. We only show how to support $\texttt{insert}$; the support for $\texttt{delete}$ is similar.

To perform $\texttt{insert}(c, i)$, we first perform a top down traversal in T to locate the leaf, v, that represents a block containing $S[i]$ in $O(\frac{\lg n}{\lg \lg n})$ time. We then increment the recorded lengths of all the mini-blocks that contain $S[i]$. We also increment the counters U of these mini-blocks, as well as the counters of their predecessors and successors. All the mini-blocks whose counters should be incremented are located in $D(v)$, $D(v_1)$ and $D(v_2)$, where v_1 and v_2 are the leaves immediately to the left and right of v in T. We scan each doubly linked list $L_l(v)$, $L_l(v_1)$ and $L_l(v_2)$ to locate these mini-blocks. Since $D(v)$, $D(v_1)$ and $D(v_2)$ have $O(\frac{\lg n}{\lg \lg n})$ mini-blocks in total over all levels, it requires $O(\frac{\lg n}{\lg \lg n})$ to find these mini-blocks and update them.

The above process can find all these mini-blocks, as well as their starting and ending positions in S. It may be necessary to reconstruct the candidate list of these mini-blocks. Similarly to the analysis in the proof of Lemma 5, the candidate list of each of these mini-blocks can be maintained in $O(1/\alpha)$ amortized time. Since there are $O(\lg \lg n)$ levels in $D(v)$, $D(v_1)$ and $D(v_2)$ and only a constant number of mini-blocks at each level may need to be rebuilt, it requires $O((\lg \lg n)/\alpha)$ amortized time to reconstruct all of them.

An insertion may also cause a mini-block to split. As in the proof of Lemma 5, we compute the candidate list and other required information for the mini-block created as a result of the merge, and amortize the cost to the insertions that lead to the merge. The amortized cost is again $O(1)$. As there can possibly be a merge at each level of $D(v)$, it requires $O(\lg \lg n)$ amortized time to handle them. Finally, when the value of L' changes, we rebuild all the data structures designed in this section,

incurring $O(\lg \lg n)$ amortized time. Therefore, the total time required to support `insert` is $O(\frac{\lg n}{\lg \lg n} + \frac{\lg \lg n}{\alpha})$. □

Combining Lemma 1 and Lemmas 4-9, we have our main result:

Theorem 10. *A sequence of length n over an alphabet of size σ can be represented using $nH_k + o(n \lg \sigma)$ bits for any $k = o(\log_\sigma n)$ to answer range alpha-majority queries in $O(\frac{\lg n}{\alpha \lg \lg n})$ time, and to support `insert` and `delete` in $O(\frac{\lg n}{\alpha})$ amortized time.*

4   Concluding Remarks

We have described the first compressed data structure for dynamic range α-majority. We perform a multi-level decomposition of the sequence S and, for each block of S, precompute a candidate set which includes all the α-majorities of any query range of the right size that touches that block. Thus, when answering a query, we need not find a set of blocks whose union forms the query range, as Elmasry *et al.* [1] do. We note we can generalize our solution to support β-majority queries for any $\beta \in [\alpha, 1]$ in $O(\frac{\lg n}{\beta \lg \lg n})$ time with the same space and update time, where β is given as part of the query and only α is fixed and given beforehand. This type of query is more general than range α-majority queries and was only studied in the static case before [7, 9]. The details are deferred to the full version.

References

[1] A. Elmasry, M. He, J. I. Munro, and P. K. Nicholson, "Dynamic range majority data structures," *Theoretical Comp. Sci.*, vol. 647, pp. 59–73, 2016.

[2] M. Fang, N. Shivakumar, H. Garcia-Molina, R. Motwani, and J. D. Ullman, "Computing iceberg queries efficiently," in *Proc. VLDB*, 1998, pp. 299–310.

[3] E. D. Demaine, A. López-Ortiz, and J. I. Munro, "Frequency estimation of internet packet streams with limited space," in *Proc. ESA*, 2002, pp. 348–360.

[4] J. Misra and D. Gries, "Finding repeated elements," *Sci. Comp. Prog.*, vol. 2, pp. 143–152, 1982.

[5] M. Karpinski and Y. Nekrich, "Searching for frequent colors in rectangles," in *Proc. CCCG*, 2008, pp. 11–14.

[6] S. Durocher, M. He, J. I. Munro, P. K. Nicholson, and M. Skala, "Range majority in constant time and linear space," *Inf. Comp.*, vol. 222, pp. 169–179, 2013.

[7] T. Gagie, M. He, J. I. Munro, and P. K. Nicholson, "Finding frequent elements in compressed 2d arrays and strings," in *Proc. SPIRE*, 2011, pp. 295–300.

[8] T. M. Chan, S. Durocher, M. Skala, and B. T. Wilkinson, "Linear-space data structures for range minority query in arrays," *Algorithmica*, vol. 72, pp. 901–913, 2015.

[9] D. Belazzougui, T. Gagie, J. Ian Munro, G. Navarro, and Y. Nekrich, "Range majorities and minorities in arrays," *CoRR*, vol. abs/1606.04495, 2016.

[10] J. I. Munro and Y. Nekrich, "Compressed data structures for dynamic sequences," in *Proc. ESA*, 2015, pp. 891–902.

[11] R. Raman, V. Raman, and S. S. Rao, "Succinct dynamic data structures," in *Proc. WADS*, 2001, pp. 426–437.

[12] L. Arge and J. S. Vitter, "Optimal external memory interval management," *SIAM J. Comp.*, vol. 32, pp. 1488–1508, 2003.