A Constant-Time Colored Choice Dictionary with Almost Robust Iteration

Torben Hagerup
Institut für Informatik, Universität Augsburg, 86135 Augsburg, Germany
hagerup@informatik.uni-augsburg.de

Abstract

A (colored) choice dictionary is a data structure that is initialized with positive integers n and c and subsequently maintains a sequence of n elements of $\{0, \ldots, c-1\}$, called colors, under operations to inspect and to update the color in a given position and to return the position of an occurrence of a given color. Choice dictionaries are fundamental in space-efficient computing. Some applications call for the additional operation of dynamic iteration, i.e., enumeration of the positions containing a given color while the sequence of colors may change. An iteration is robust if it enumerates every position that contains the relevant color throughout the iteration but never enumerates a position more than once or when it does not contain the color in question. We describe the first choice dictionary that executes every operation in constant amortized time and almost robust iteration in constant amortized time per element enumerated. The iteration is robust, except that it may enumerate some elements a second time. The data structure occupies $n \log_2 c + O((\log n)^2)$ bits. The time and space bounds assume that $c = O((\log n)^{1/2}(\log\log n)^{-3/2})$.

2012 ACM Subject Classification Theory of computation → Data structures design and analysis

Keywords and phrases Succinct data structures, space efficiency, in-place chain technique, bounded universes, amortization

Digital Object Identifier 10.4230/LIPIcs.MFCS.2019.64

1 Introduction

Following similar earlier definitions [3, 5] and concurrently with that of [2], the (colored) choice dictionary was introduced by Hagerup and Kammer [10]. It appears to be fundamental in space-efficient computing and has already been shown to have a number of applications [2, 5, 10, 11, 12]. Its precise definition is as follows:

▶ **Definition 1.1.** *A (colored) choice dictionary is a data structure that can be initialized with integers $n, c \in \mathbb{N} = \{1, 2, \ldots\}$ and subsequently maintains a sequence $(e_1, \ldots, e_n)$ of n integers drawn from $\{0, \ldots, c-1\}$, initially $(0, \ldots, 0)$, under the following operations:*

color(i) $(i \in \{1, \ldots, n\})$: *Returns e_i.*
setcolor(j, i) $(j \in \{0, \ldots, c-1\}$ *and* $i \in \{1, \ldots, n\})$: *Replaces e_i by j.*
choice(j) $(j \in \{0, \ldots, c-1\})$: *With $S_j = \{i \in \{1, \ldots, n\} : e_i = j\}$, returns an (arbitrary) element of S_j if $S_j \neq \emptyset$, and 0 if $S_j = \emptyset$.*

We call the elements of $\{0, \ldots, c-1\}$ *colors*. The choice dictionary is similar to an array of colors that supports reading (*color*) and writing (*setcolor*), but with a crucial additional operation (*choice*) to locate an occurrence of a given color. Our terminology will sometimes pretend that the elements of $\{1, \ldots, n\}$ have colors. For $j = 0, \ldots, c-1$, we define $S_j = \{i \in \{1, \ldots, n\} : e_i = j\}$ as above and call S_j a *color class.*

In some applications of choice dictionaries, notably to the computation of a breadth-first search or BFS forest of a given graph [9, 10], it is essential for a choice dictionary to support the additional operation of iteration over a given color class (while other computation takes place in an interleaved fashion). The BFS algorithms of [9, 10], for instance, repeatedly

44th International Symposium on Mathematical Foundations of Computer Science (MFCS 2019).
Editors: Peter Rossmanith, Pinar Heggernes, and Joost-Pieter Katoen; Article No. 64; pp. 64:1–64:14
Leibniz International Proceedings in Informatics
LIPICS Schloss Dagstuhl – Leibniz-Zentrum für Informatik, Dagstuhl Publishing, Germany

iterate over the *gray* vertices, those with a given distance d from a start vertex, in order to identify the vertices at distance $d + 1$. This is not straightforward because the algorithms – in order to save colors – also color the vertices at distance $d + 1$ gray. In other words, the set of gray vertices changes while it is being iterated over. Hagerup and Kammer [10] coined the phrase *robust iteration* to describe the ideal that one would like to have in such a situation. An iteration over a dynamic set S is robust if every element that belongs to S throughout the iteration is certain to be enumerated by the iteration, while no element is enumerated more than once or at a time when it does not belong to S. Elements that belong to S during part but not all of the iteration may or may not be enumerated. We say that an iteration works in constant time if there are constant-time operations to initialize for a new iteration, to test whether more elements remain to be enumerated, and – if so – to enumerate the next element.

By the information-theoretic lower bound, a choice dictionary with parameters n and c must occupy at least $\lceil n \log c \rceil$ bits ("log" always denotes the binary logarithm function $\log_2$). Kammer and Sajenko [13] recently described a choice dictionary that occupies $n \log c + O(\log n)$ bits and executes every operation including iteration in constant time, but is severely restricted:

- The number c of colors must be a power of 2.
- Moreover, c must be a constant.
- The iteration is *static*: While an iteration is underway, no changes to colors are allowed.

We present a new choice dictionary that also executes every operation including iteration in constant time and alleviates the drawbacks listed above. The number of colors is not restricted to be a power of 2, it is not required to be a constant – but we do impose the condition $c = O((\log n)^{1/2}(\log\log n)^{-3/2})$ – and the iteration is dynamic and almost robust. By "almost robust" we mean that the definition of robust iteration is satisfied, except that some elements may be enumerated a second time (the BFS algorithms discussed above can tolerate this). Compared to the choice dictionary of [13], the new data structure has a few drawbacks of its own:

- It is more complicated.
- The operation times are amortized, not worst-case.
- The number of bits needed is $n \log c + O((\log n)^2)$, not $n \log c + O(\log n)$.

The third drawback should be seen in light of the fact that even if only the operations *color* and *setcolor* (i.e., those of an array) are to be supported in constant time, every known data structure that does not restrict c to be a power of 2 requires $n \log c + \Omega((\log n)^2)$ bits. As for the condition $c = O((\log n)^{1/2}(\log\log n)^{-3/2})$, it may be noted that the performance of comparable choice dictionaries [7, 10] also degrades sharply if the number of colors exceeds similar thresholds.

The main novelty of our work lies in the almost robust iteration. Allowing amortization changes the ground rules of iteration and was crucial to obtaining the results described here. It becomes feasible to begin a dynamic iteration over a color class S_j by an "internal" static iteration that can serve, among other things, to determine $|S_j|$. Enumerating the elements that belong to S_j at that time, i.e., at the start of the iteration, would satisfy the conditions of robustness, but the elements that are enumerated must be handed to a caller one by one, and there is no space to store them temporarily (cf. the BFS algorithms discussed above). Once S_j is allowed to change while it is being iterated over, it becomes difficult to keep track of which elements have already been enumerated and therefore to prevent elements from being enumerated repeatedly and perhaps an unbounded number of times (i.e., the iteration might not terminate). Enumerating the elements of S_j in sorted order would take care of this problem, but constant time per element is not enough to sort. Our iteration sorts a small part of S_j initially. If the changes to S_j during the iteration are relatively few, the partial sorting is sufficient to guarantee robustness. If there are many changes to S_j, on the other hand, these can "pay for" a complete sorting of what remains of S_j. An element is enumerated at most once before and at most once after the complete sorting.

The only previous choice dictionary with robust iteration was devised by Hagerup and Kammer [10], but for worse time and/or space bounds (e.g., constant time together with $n \log c + O(n/(\log n)^t)$ bits for constant c and arbitrary fixed $t \in \mathbb{N}$). It should be noted that in contrast with the robust iteration of [10], if the iteration described here is terminated early, its time bound is the same as if it had run to completion, i.e., incomplete iteration is not supported efficiently.

2 Preliminaries

Our model of computation is the standard word RAM [1, 6] with a word length of $w \in \mathbb{N}$ bits, where we assume that w is large enough to allow all memory words in use to be addressed. As part of ensuring this, we assume that $w \geq \log n$. The word RAM has constant-time operations for addition, subtraction and multiplication modulo 2^w, division with truncation ($(x, y) \mapsto \lfloor x/y \rfloor$ for $y > 0$), left shift modulo 2^w ($(x, y) \mapsto (x \ll y) \bmod 2^w$, where $x \ll y = x \cdot 2^y$), right shift ($(x, y) \mapsto x \gg y = \lfloor x/2^y \rfloor$), and bitwise Boolean operations (AND, OR and XOR (exclusive or)). The machine is also assumed to be able to compute $\lfloor \log x \rfloor$ in constant time for every given $x \in \{1, \ldots, 2^w - 1\}$.

Like all comparable data structures, the new choice dictionary depends on standard low-level word-RAM routines, some of which are conveniently collected in the following lemma.

▶ **Lemma 2.1** ([10], Lemma 3.2). *Let m and f be given integers with $1 \leq m, f < 2^w$ and suppose that a sequence $A = (a_1, \ldots, a_m)$ with $a_i \in \{0, \ldots, 2^f - 1\}$ for $i = 1, \ldots, m$ is given in the form of the (mf)-bit binary representation of the integer $\sum_{i=0}^{m-1} 2^{if} a_{i+1}$. Then the following holds:*

(a) *Let $I_0 = \{i \in \{1, \ldots, m\} : a_i = 0\}$. Then, in $O(1 + mf/w)$ time, we can test whether $I_0 = \emptyset$ and, if not, compute* $\min I_0$.

(b) *If an additional integer $k \in \{0, \ldots, 2^f - 1\}$ is given, then $O(1 + mf/w)$ time suffices to compute the integer $\sum_{i=0}^{m-1} 2^{if} b_{i+1}$, where $b_i = 1$ if $k \geq a_i$ and $b_i = 0$ otherwise for $i = 1, \ldots, m$.*

(c) *If $m < 2^f$ and an additional integer $k \in \{0, \ldots, 2^f - 1\}$ is given, then $\mathit{rank}(k, A) = |\{i \in \{1, \ldots, m\} : k \geq a_i\}|$ can be computed in $O(1 + mf/w)$ time.*

3 Informal Overview

This section tries to convey the basic intuition behind the new data structure. The necessary nitty-gritty details and calculations will be presented in the subsequent sections.

It is known from the work of Dodis, Pătraşcu and Thorup [4] that a sequence of n color values drawn from $\{0, \ldots, c - 1\}$ can be stored in a data structure that occupies $n \log c + O((\log n)^2)$ bits and supports the operations *color* and *setcolor* in constant time. Let us call this structure a *c-ary array*. Our task can be viewed as that of adding efficient *choice* and almost robust iteration to a c-ary array.

Suppose that, in addition to being represented in a c-ary array, the elements of each color class S_j, where $j \in \{0, \ldots, c-1\}$, are organized in a doubly-linked list L_j. Then executing *choice*(j) is trivial – return an arbitrary element of L_j – and with a little care we can also carry out a robust iteration over S_j via a traversal of L_j during which elements that drop out of S_j splice themselves out of L_j and elements that enter S_j are inserted at the beginning of L_j, where they will not be enumerated – they may be "trying to sneak in again" after having already been enumerated in the past.

The problem with the scheme outlined above is, of course, that the doubly-linked lists $L_0, \ldots, L_{c-1}$ would need far too much space – $\Theta(n \log n)$ bits – for two pointers for each of the n elements. In order to alleviate this problem, we divide the n elements into equal-sized groups and represent each group via a data structure called a *container.* Provided that groups are not too large, iteration within a container can be handled efficiently with techniques based on Lemma 2.1. If every container holds at least one element of a color class S_j, we can therefore iterate over S_j by iterating over all containers and enumerating the elements of S_j in each. Thus at this point the problem can be viewed as represented by the containers that are *j-free*, i.e., do not contain any occurrences of j. The list L_j that we would now like to have for each $j \in \{0, \ldots, c-1\}$ chains together those containers that are not j-free. In contrast with what was the case before, a container can belong to many lists, for which reason it may need up to $2c$ pointers rather than just two pointers.

Even though we have reduced the need for pointers from two per element to $2c$ per container, the basic problem remains that we have already used up practically all the available space and cannot afford to store even a single pointer per container. If some color is missing entirely from a container, however, $\log c$ bits per element is a tad more than what is strictly necessary to record the contents of the container. Using a more efficient encoding, we can compress the information stored in the container slightly to leave room for a couple of pointers – provided that containers are sufficiently large. The same is true if a color is not missing completely, but has very few occurrences, as we can then store the sequence of these "exceptional" occurrences explicitly while using the efficient encoding for the other elements in the container. If several colors are missing or rare in a container, we can pack the container even more tightly. As a result, the container has room for a couple of pointers for each such color, which in turn means that we can treat the colors somewhat independently.

Let $j \in \{0, \ldots, c-1\}$ and let us call a container H *j-sparse* if the number of occurrences of j in H is bounded by some suitably chosen r, and *j-abundant* otherwise. For a moment, let us make the unrealistic assumption that all j-sparse containers are to the left of (have smaller indices than) all j-abundant containers. Then we can keep in L_j those containers that are j-sparse (and therefore have room for the necessary pointers) but not j-free (since a main goal is to skip the j-free containers). To iterate over S_j, iterate both over L_j and over the j-abundant containers. The latter is easy because the j-abundant containers, by assumption, are consecutively numbered.

Even though the unrealistic assumption is unrealistic, we can still keep track of the number μ_j of j-sparse containers and imagine that the leftmost μ_j containers (say that these form the *left part*) "normally" are j-sparse. There may be containers that go against the norm, j-abundant containers in the left part – call these *j-masters* – and j-sparse containers in the right part, *j-slaves.* It is easy to see that the number of j-masters equals the number of j-slaves, and the *in-place chain technique* of Katoh and Goto [14] suggests to maintain a perfect matching between the j-masters and the j-slaves. This helps us to solve two problems: (1) A j-master, by virtue of not being in the right part, needs to belong to L_j in order to be iterated over, but has no room for pointers. (2) Because it is in the right part, a j-slave is iterated over, but may be j-free, in which case it cannot "pay for" its share of the iteration. To solve problem (1), we relocate some of the data of the master to the slave, which has room to spare. As for problem (2), even though the slave cannot "pay", its master can.

We cannot keep track of the number of occurrences of a color j in a container once that number exceeds r, and therefore replace the notions j-sparse and j-abundant by algorithmically more tractable *j-weak* and *j-strong*. A j-weak container is certain to be j-sparse, but a j-strong container may be j-sparse or j-abundant. If the number of occurrences of j in a j-weak container H grows beyond r, H is converted to being j-strong. A conversion of H back to j-weak, however, takes place only if and when H is later discovered to have fewer than r occurrences of j, and this in turn happens essentially only in connection with an iteration over H. The iteration can "pay" a constant for each element that is enumerated, and the operations that changed the colors of the other elements formerly of color j can "pay" a constant for each of these. In an amortized setting, therefore, a conversion between j-weak and j-strong can be allowed to have a cost of $O(r)$, and r is chosen accordingly. A budget of $O(r)$ also covers the conversion of a j-strong container, for the purpose of iteration over the container, from the very compact usual representation to one that supports efficient iteration.

A final problem is represented by containers that migrate from the left to the right side (as a consequence of a decrease in μ_j). Such a container might be iterated over on the left side, because it belongs to L_j, and later again as a container on the right side. In order to prevent elements from being enumerated repeatedly, the iteration is designed to enumerate elements roughly in sorted order. In more detail, first the number $|L_j|$ of containers in L_j is determined, and then the containers in L_j that belong to a "buffer zone" of width $|L_j|^2$ at the right end of the left part are sorted, which can happen in $O(|L_j|+1)$ time, and placed last in L_j. If the buffer zone is consumed entirely, i.e., migrates completely to the right part before it starts to take part in the iteration, this is evidence of so many color changes during the iteration that we can afford to sort the remaining containers in L_j. After that point in time, an element is enumerated at most once, but it may also have been enumerated once before the sorting. If some part of the buffer zone remains in the left part throughout the iteration over the part of L_j on its left, on the other hand, the order of iteration over the containers on its left is immaterial, and the iteration is robust.

4 The Data Representation

This section describes the organization of data in the choice dictionary after it has been initialized with parameters $n, c \ge 2$ with $c = O((\log n)^{1/2}(\log\log n)^{-3/2})$. We first describe a slightly bigger data structure that uses $n \log c + O(c(\log n)^2)$ bits.

For a positive integer $s = \Theta(c \log n)$, we divide the sequence $(e_1, \ldots, e_n)$ of n color values to be maintained into $N = \lfloor n/s \rfloor$ subsequences of exactly s consecutive color values each, with at most $s-1$ color values left over. The left-over color values can be handled separately in $O(s \log s) = O((\log n)^2)$ bits, essentially by keeping a doubly-linked list for each color of the positions in which the color occurs. We omit the easy details and assume in the following that n is a multiple of s. For $\ell = 1, \ldots, N$, the ℓth subsequence is stored in a data structure H_ℓ called a *container* (as anticipated in the previous section). Each container partitions the set $\{0, \ldots, c-1\}$ of colors into a set of *weak* colors and the complementary set of *strong* colors. If a color $j \in \{0, \ldots, c-1\}$ is weak in a container H, we call H *j-weak*; otherwise H is *j-strong* (again, these terms as well as notation introduced below were used already in Section 3). For $j = 0, \ldots, c-1$, we keep a global count $\mu_j \in \{0, \ldots, N\}$ of the number of j-weak containers. For $\ell \in \{1,\ldots,N\}$ and $j \in \{0,\ldots,c-1\}$, we say that H_ℓ is *j-left* if $\ell \le \mu_j$ and *j-right* if $\ell > \mu_j$. Finally, we call H_ℓ *j-free* if the sequence of color values stored in H_ℓ does not contain any occurrence of j.

Let $\ell \in \{1,\ldots,N\}$ and suppose that $(a_0,\ldots,a_{s-1})$ is the subsequence of color values stored in H_ℓ – thus $a_i = e_{(\ell-1)s+i+1}$ for $i = 0,\ldots,s-1$. If H_ℓ is j-weak, where $j \in \{0,\ldots,c-1\}$, we require the number $|\{i \in \{0,\ldots,s-1\} : a_i = j\}|$ of occurrences of j in H_ℓ to be bounded by a positive integer r with $r = \Theta(\log n/(c\log\log n))$.

If H_ℓ is j-strong and j-right for all $j \in \{0,\ldots,c-1\}$, H_ℓ is *atomic*, by which we mean that its representation, which we will call $D_{\overline{W}}$, consists of the single integer $x = \sum_{i=0}^{s-1} a_i c^i$. Provided that we keep a table of the powers $c^0, c^1, \ldots, c^{s-1}$, we can then inspect and update individual values in $(a_0,\ldots,a_{s-1})$ with a constant number of arithmetic operations. E.g., $a_i = \lfloor x/c^i \rfloor \bmod c$ for $i = 0,\ldots,s-1$. This does not necessarily translate into constant time, as we may not be able to operate on integers as large as c^s in constant time. To remedy this, we obtain s as $s_0 s_1$, where s_0 and s_1 are positive integers with $s_0 = \Theta(\log n/\log c)$ and $s_1 = \Theta(c\log c)$, and actually represent x through s_1 digits – called *big digits* – to base $C = c^{s_0} = n^{O(1)}$. We can operate on big digits in constant time, and it is easy to see that we can still inspect and update individual color values by operating only on the appropriate big digit. Now we need only the powers $c^0,\ldots,c^{s_0-1}$. For reasons that will become clear shortly, we store in fact a table of all powers bounded by C of the form i^k, where $i \in \{2,\ldots,c\}$ and $k \in \mathbb{N}$. The number of bits required for the table is easily seen to be $O(c(\log n)^2)$.

In all other cases, namely if H_ℓ is j-weak or j-left for at least one $j \in \{0,\ldots,c-1\}$, H_ℓ is *composite*, namely represented through a collection of four substructures: D_{C}, which stores information concerning the roles played by the various colors in H_ℓ; D_W and $D_{\overline{W}}$, which specify the distribution of weak and strong colors in H_ℓ, respectively; and D_{P}, which links H_ℓ to other containers. The details are as follows:

- First, for various subsets M of $\{0,\ldots,c-1\}$ we store $\mathit{rank}_M(j) = |\{i \in M : i \le j\}|$ for $j = 0,\ldots,c-1$ and $\mathit{select}_M(k) = \min\{j \in \mathbb{N}\cup\{0\} : \mathit{rank}_M(j) = k\}$ for $k = 1,\ldots,|M|$. Thus we store a table of rank_M, which numbers the elements of M consecutively, and a table of select_M, which realizes the inverse mapping. In detail, let W, $\overline{W}$, R and $\overline{R}$ be the sets of colors j for which H_ℓ is j-weak, j-strong, j-right and j-left, respectively, Then we store tables of *rank* and *select* for all of the sets W, $\overline{W}$, R and $\overline{R}$ as well as for all unions and intersections of two of these (we actually need only a few of the tables). D_{C} is the collection of these tables, which occupy $O(c\log c)$ bits and can be computed from W and R in $O(c)$ time.
- Second, D_W stores the set $\{(i,a_i) \mid i \in \{0,\ldots,s-1\}$ and $a_i \in W\}$ and, more abstractly, maintains a subset A of $\{0,\ldots,s-1\}\times\{0,\ldots,c-1\}$ and supports the following operations in constant time:
  - Given a pair $(i,j) \in A$, delete it from A.
  - Given a pair $(i,j) \in \{0,\ldots,s-1\}\times\{0,\ldots,c-1\}$, insert it in A, provided that before the operation no pair in A has first component i and fewer than r pairs in A have second component j.
  - Given $i \in \{0,\ldots,s-1\}$, return the pair of the form (i,j) in A or an indication that A contains no such pair.
  - Given $(i,j) \in \{0,\ldots,s-1\}\times\{0,\ldots,c-1\}$, return the pair of the form (i',j) in A with $i' \ge i$ for which i' is minimal or an indication that A contains no such pair.
  - Given $j \in \{0,\ldots,c-1\}$, return the number of pairs of the form (i,j) in A.

  D_W can be thought of as an embellished associative array for the weak colors. What makes its realization easy is the fact that the number of pairs in A at all times is bounded by cr, while every component of a pair is an integer of f bits, where $f = O(\log s) = O(\log\log n)$. We can simply store $|A|$, an integer of $O(\log(cr)) = O(\log\log n)$ bits, and two arrays A_1 and A_2 of cr f-bit entries each such that the first $|A|$ entries in A_1 contain the first components of the pairs in A in sorted order and the first $|A|$ entries in A_2 contain their second components in the corresponding order, i.e., so that the two components of each pair have the same index in A_1 and A_2. Each of A_1 and A_2 is stored as a single integer of $crf = O(\log n)$ bits, and without loss of generality (choose s_0 and hence s sufficiently large) we will assume that D_{C} and D_W fit together in a single big digit.

To delete a pair (i, j) from A, we can form the bitwise XOR of A_1 with an integer that contains the integer i in each of $|A|$ f-bit fields and can be obtained in constant time with a multiplication, and subsequently use the algorithm of Lemma 2.1(a) to discover the position in A_1 and A_2 of the pair (i, j) (or an indication that (i, j) does not occur in A). After subtracting 1 from the variable that stores $|A|$, we can easily carry out the remainder of the operation, namely prizing out a field from A_1 and A_2 and closing the gap, in constant time with a combination of bitwise Boolean operations, applications of bit masks and shifts.

When inserting a pair (i, j) in A, rather than an exact match of i in A_1, we want to find the correct place in which to insert i while keeping A_1 sorted. We therefore use part (c) instead of part (a) of Lemma 2.1, but can otherwise proceed similarly as in the case of a deletion. The remaining operations listed above can be implemented to work in constant time in similar ways and are left to the interested reader (the last operation can also be handled by maintaining an additional array of counts).

- Third, the distribution of strong colors in H_ℓ is recorded in $D_{\overline{W}}$ in one of several formats, chosen in dependence of $|\overline{W}|$ and c: If $|\overline{W}| = 0$, there are no strong colors in H_ℓ, and $D_{\overline{W}}$ is a dummy data structure (denoted by $\emptyset$, say) that takes up no space. If $|\overline{W}| = 1$ and $c = 2$, we also take $D_{\overline{W}} = \emptyset$, since the necessary information can be gleaned from D_W – an element whose color is stored in H_ℓ has the strong color precisely if it does not have the weak color. If $1 \le |\overline{W}| \le c - 2$, $D_{\overline{W}}$ stores the sequence $(a'_0, \ldots, a'_{s-1})$, where $a'_i = 0$ if $a_i \in W$ and $a'_i = \mathit{rank}_{\overline{W}}(a_i)$ if $a_i \in \overline{W}$, for $i = 0, \ldots, s - 1$, essentially in the form of the integer $y = \sum_{i=0}^{s-1} a'_i(|\overline{W}| + 1)^i$. In other words, every weak color is encoded via a zero, the strong colors are encoded via the integers $1, \ldots, |\overline{W}|$, and the sequence of codes is represented similarly as for atomic containers. Finally, if $|\overline{W}| \ge \max\{c - 1, 2\}$, we redefine a'_i as $\mathit{rank}_{\overline{W}}(a_i) - 1$ for $a_i \in \overline{W}$ and store $y = \sum_{i=0}^{s-1} a'_i |\overline{W}|^i$. Thus for $|\overline{W}| = c - 1 \ge 2$ we give up on distinguishing between the weak color and one strong color.

Informally, the idea behind D_W and $D_{\overline{W}}$ is that weak colors have only few occurrences, which can be stored in little space by listing them explicitly (in D_W), while this allows the strong colors to be represented more economically via smaller codes (in $D_{\overline{W}}$). Having two distinct conventions for y is a necessary complication: We must be able to store H_ℓ in less space even if only a single color is weak, which precludes the use of the first convention (i.e., reserving a code value for weak colors) when $|\overline{W}| \ge c - 1$. On the other hand, the second convention (not distinguishing between weak colors and the smallest strong color j_0) cannot be used when $|W| = c - |\overline{W}|$ is large, since it requires us to find the occurrences of j_0 by testing all occurrences of a zero in $(a'_0, \ldots, a'_{s-1})$ and filtering out those that correspond to weak colors – we cannot allow too many "false positives". Moreover, the case $|\overline{W}| = c - 1$ must be handled separately in $D_{\overline{W}}$ for $c = 2$ (namely not at all), because $|\overline{W}| = 1$ cannot be used as the basis of a positional system.

As was the case for x, if $1 \le |\overline{W}| \le c - 2$ or $|\overline{W}| \ge \max\{c - 1, 2\}$ it is necessary to express y through a sequence of big digits but, unless $|\overline{W}| = c$, the task is now hampered by rounding issues. Assume that $|\overline{W}| \le c - 1$. The s *small digits* to base $|\overline{W}| + h$, where $h = 1$

if $1 \le |\overline{W}| \le c-2$ and $h=0$ if $|\overline{W}| = c-1 \ge 2$, are to be distributed over a number of big digits to base $C = c^{s_0}$. A single big digit can accommodate $\lfloor (s_0 \ln c)/\ln(|\overline{W}|+h)\rfloor$ small digits. Now $\ln(|\overline{W}|+h) = \ln(c-|W|+h) = \ln c + \ln(1-(|W|-h)/c) \le \ln c - (|W|-h)/c \le \ln c - |W|/(2c)$ and

$$\frac{s_0 \ln c}{\ln(|\overline{W}|+h)} \ge \frac{s_0 \ln c}{\ln c - |W|/(2c)} = \frac{s_0}{1-|W|/(2c\ln c)} \ge s_0(1+|W|/(2c\ln c)).$$

Since $|W| \ge 1$ and $s_0/(c\ln c) = \Omega(\ln n/(c(\ln c)^2)) = \omega(1)$, we may conclude that a big digit can accommodate $s_0(1+\Omega(|W|/(c\ln c)))$ small digits (even taking the $\lfloor\cdots\rfloor$ operation into account). Moreover, since

$$\frac{s}{s_0(1+\Omega(|W|/(c\ln c)))} = \frac{s_1}{1+\Omega(|W|/(c\ln c))} = s_1(1-\Omega(|W|/(c\ln c)))$$

and $s_1 = \Theta(c\ln c)$, by choosing s_1 and hence s sufficiently large we can ensure that the total number of big digits necessary to accommodate all s small digits is at most $s_1 - 2K|W|$ for an arbitrary constant $K \in \mathbb{N}$ of our choice (but independent of s).

- Fourth and finally, D_{P} is an array that maps each color $j \in \{0,\ldots,c-1\}$ for which H_ℓ is j-weak or j-left to a *block* of three pointers, each of which points to a container or has the value *null* (points to nothing). By choosing s_0 and therefore s sufficiently large, we can assume that a block fits in a big digit. Two of the pointers stored for a color j are used to organize certain j-left containers in a doubly-linked list L_j, namely those that are j-strong or not j-free (informally, those that might contain occurrences of j). The third pointer, called a *matching pointer*, is used (has a value different from *null*) only if H_ℓ is j-left and j-strong – then H_ℓ is called a *j-master* – or H_ℓ is j-right and j-weak – then H_ℓ is called a *j-slave*. As essentially noted before, the number of j-masters always equals the number of j-slaves, and the matching pointers form a perfect matching between the j-masters and the j-slaves, with each matching pointer of a master pointing to its corresponding slave, and vice versa.

In order to reduce the space needed for tables of powers of integers to $O((\log n)^2)$ bits, we replace the set $\{c, c-1, \ldots, 2\}$ of possible bases used to represent the integer y of $D_{\overline{W}}$ by the smaller set $\{c, c-1, c-2, c-4, c-8, \ldots\}$: The base $c-i$, where $i \in \{1,\ldots,c-2\}$, is replaced by $c - 2^{\lfloor \log i \rfloor}$. Then y takes up more space, but the inequality $2^{\lfloor \log(|W|-h)\rfloor} \ge |W|/2$ for $0 \le h \le \min\{|W|-1, 1\}$ shows that it is still the case that y can be represented in $s_1 - 2K|W|$ big digits, where W is the set of weak colors of the container under consideration. Now we need powers of only $O(\log c)$ integers, at most one of which is smaller than $\sqrt{c}$, so the number of bits needed comes to $O(\log c \cdot s_0 \log C) = O((\log n)^2)$. This ends the description of the structure of a composite container H_ℓ.

The use of masters and slaves is an element of the in-place chain technique of Katoh and Goto [14]. Another element requires us to distinguish between *conceptual containers* (intended until this point) and *physical containers*. The number of big digits needed to store a composite container H_ℓ was bounded above by $1 + (s_1 - 2K|W|) + |W \cup \overline{R}|$, where W and $\overline{R}$ are the sets of colors j for which H_ℓ is j-weak and j-left, respectively. This number varies from one (conceptual) container to another, which is precisely the problem – we cannot store the containers in little space so as to allow constant-time access to a given container. This problem is solved by a transfer of data between containers. For each $j \in \{0,\ldots,c-1\}$ and each pair (H, H') of containers such that H and H' are a j-master and its j-slave, respectively, K big digits are relocated from H to H'. A container chooses the at most Kc big digits to relocate as a master to be among at least $s_1 - 1 - c = \omega(c)$ big digits reserved for $D_{\overline{W}}$ and not to overlap with the at most Kc digits received as a slave – if s_1 and hence s are chosen sufficiently large, this is always possible. The first condition means that no "bookkeeping information" is moved to a place where it might be difficult to find (most obviously, the pointer to a slave should not be relocated to the slave), and the second condition ensures that a master can always access one of its relocated big digits in constant time – it must follow a pointer chain of length 1 only.

With the transfer of data described above, the number of big digits needed by a composite container H_ℓ changes to

$$\begin{aligned}&1 + (s_1 - 2K|W|) + |W \cup \overline{R}| - K|\overline{W} \cap \overline{R}| + K|W \cap R|\\ \le\ &1 + s_1 - K(|W| + |\overline{W} \cap \overline{R}|) + |W \cup \overline{R}|\\ =\ &s_1 + 1 - (K-1)|W \cup \overline{R}|\end{aligned}$$

where, again, W, $\overline{W}$, R and $\overline{R}$ are those of H_ℓ. Because H_ℓ is composite, $|W \cup \overline{R}| \ge 1$, so that the quantity above is at most $s_1 - (K-2)$. After the transfer of data, therefore, every conceptual container H can be stored in a physical container of exactly s_1 big digits, and if H is composite, it additionally offers $K-2$ big digits of freely usable *extra space.*

The $P = Ns_1$ big digits of the N physical containers are maintained in an instance of the ingenious data structure of Dodis et al. [4] embodied in the lemma below. The number of bits needed is $P \log C + O((\log P)^2) = Ns_1 s_0 \log c + O((\log(Ns_1))^2) = n \log c + O((\log n)^2)$.

▶ **Lemma 4.1** ([4], Theorem 1)**.** *There is a data structure that, given arbitrary positive integers P and C with $C = P^{O(1)}$, can be initialized in $O(\log P)$ time and subsequently maintains an array of P elements drawn from $\{0, \ldots, C-1\}$ in $P \log C + O((\log P)^2)$ bits such that individual array elements can be read and written in constant time.*

When nothing else is stated, in the following "container" means "conceptual container". A subtle point is that an atomic container must be prevented from "posing as" a composite container – there is no space for an atomic/composite bit, and every bit combination is in use for an atomic container. If an atomic container H "claims" to be j-left for some $j \in \{0, \ldots, c-1\}$, such a j can be found in constant time with D_{C}, and the claim can be falsified in constant time by inspection of μ_j. Otherwise, if H "claims" to be j-weak for some $j \in \{0, \ldots, c-1\}$ and hence a j-slave, the claim can be falsified in constant time by the lack of a reverse pointer to H in its purported j-master. Thus we can always test in constant time whether a given container is atomic or composite.

In addition to the "per-container" data detailed above, we store globally, for each color, a *choice buffer* and an *iteration buffer*, maintained in a special *loose representation* that is wasteful of space, but allows efficient operations corresponding to *choice* and robust iteration. Each buffer is *derived* from a container H_ℓ whose index ℓ is remembered. If the data structure $D_{\overline{W}}$ of H_ℓ stores the sequence $(a'_0, \ldots, a'_{s-1})$ of codes, the loose representation of H_ℓ consists of the integer $\sum_{i=0}^{s-1} 2^{if} a'_i$, where $f = \lceil \log c \rceil$. The derivation of a choice or iteration buffer can be carried out in $O(\log s_0)$ time per big integer and $O(s_1 \log s_0) = O(c \log c \log \log n) = O(r)$ time altogether with the algorithm of Lemma 4.2 below, which is a word-parallel version (i.e., essentially independent computations take place simultaneously in different regions of a word) of a simple divide-and-conquer procedure.

▶ **Lemma 4.2** ([7], Lemma 3.3 with $p = 1$)**.** *Given positive integers c, d, f and s with $c, d \ge 2$ and $f \ge \lceil \log_2 \max\{c, d\} \rceil$ and an integer of the form $\sum_{j=0}^{s-1} a_j c^j$, where $0 \le a_j < \min\{c, d\}$ for $j = 0, \ldots, s-1$, the integer $\sum_{j=0}^{s-1} a_j d^j$ can be computed in $O(\lceil sf/w \rceil (\log s + (\log(2 + sf/w))^2))$ time.*

Because each code a'_i is readily available in an f-bit field in the loose representation, we can use the algorithm of Lemma 2.1(a), applied to a buffer for a color j and derived from a container H, to carry out a *sweep* of H that reports all elements stored in H whose color is j when they are hit by the sweep. This takes $O(1 + sf/w) = O(c \log c)$ time plus time proportional to the number of occurrences reported plus, possibly, $O(r)$ time to skip occurrences of weak colors with the same code as j, and clearly satisfies the conditions for a robust iteration over H. We generally suspend the sweep as soon as we have found one occurrence of j and resume the sweep later to find the next occurrence, for which reason we also store with each buffer how far the current sweep over the buffer has progressed. In the case of a choice buffer, the sweep is resumed one position earlier so that it reports the same element as last time unless the color of that element has changed. The space needed is $O(sf) = O(c \log c \log n)$ bits per buffer and $O(c^2 \log c \log n) = O((\log n)^2)$ bits for all $2c$ buffers.

When a buffer for a color j derived from a container H is replaced by a different buffer, j is made weak in H if this is possible, i.e., if j occurs at most r times in H. The procedure for doing this is described in the following section.

Initially all containers are 0-strong and j-weak for $j = 1, \ldots, c-1$, and there are no masters or slaves.

5 Conversion Between Weak and Strong Colors

If the number of occurrences of a color j in a container H_ℓ is bounded by r, H_ℓ may be changed from j-weak to j-strong, or vice versa. This section describes the details.

Assume first that H_ℓ is composite both before and after the change. Let the sets W, R, etc., and the substructures D_{C}, D_W, etc., be those of H_ℓ. First, W changes in the obvious way by gaining or losing an element, and D_{C} is recomputed accordingly from skratch in $O(c)$ time. Second, up to r pairs are inserted in or removed from D_W, which can happen in $O(r)$ time.

Consider now the necessary update of $D_{\overline{W}}$. Recall that in the nontrivial cases, $D_{\overline{W}}$ is represented by an integer y composed of s small digits to base $|\overline{W}|$ or $|\overline{W}| + 1$ distributed over a number of big digits. Using the algorithm of Lemma 4.2, we first convert y to the loose representation, which takes $O(r)$ time as before. Informally, we must either create a gap among the code values of the existing strong colors to make room for a new strong color or, conversely, prize out the code value of a color that stops being strong and subsequently close the gap that it leaves. Both of these can be done in $O(r)$ time with the algorithm of Lemma 2.1(b). We omit the details. For $c \geq 3$, these vary a little depending on whether or not $|\overline{W}|$ switches between $c-2$ and $c-1$, i.e., depending on whether or not the interpretation of y switches between the first and the second convention. After the creation of a gap for the code value of a new strong color, the $r+1$ occurrences of that code are "planted" one by one in $O(r)$ time. Similarly, the occurrences of the code of a color that becomes weak are replaced by the code of weak colors (i.e., zero) before the code value of the formerly strong color is prized out. When these changes have taken place, the algorithm of Lemma 4.2 is applied again to convert the loose representation back to the usual representation of $D_{\overline{W}}$ as an integer y. Altogether, the update of $D_{\overline{W}}$ can happen in $O(r)$ time.

The final task is to repair the matching between j-masters and j-slaves. In the special case in which H_ℓ switches not only between j-weak and j-strong, but simultaneously between j-left and j-right (this happens if μ_j switches between ℓ and $\ell - 1$), H_ℓ is neither a j-master nor a j-slave neither before nor after the switch, and nothing needs to be done. Otherwise let $H \neq H_\ell$ be the container that switches as above between j-left and j-right because of the change in μ_j. There are two main cases: (1) If H_ℓ switches from j-weak to j-strong and therefore μ_j decreases by 1 and H switches from j-left to j-right, H_ℓ either becomes a j-master (it is j-left) or stops being a j-slave (it is j-right), and H either stops being a j-master (it is j-strong) or becomes a j-slave (it is j-weak). In all combinations there is a master without a slave and a slave without a master, and they are matched. (2) If H_ℓ switches from j-strong to j-weak and H switches from j-right to j-left, the situation is completely analogous: H_ℓ either becomes a j-slave (it is j-right) or stops being a j-master (it is j-left), and H either stops being a j-slave (it is j-weak) or becomes a j-master (it is j-strong). Again, there is a slave without a master and a master without a slave, and they are matched. In each case the matching pointers are adjusted and the data relocated from masters to slaves is moved appropriately. In addition, the substructure D_{C} of H must be updated. All of this can be done in $O((s \log c)/w) = O(r)$ time.

If H_ℓ is atomic either before or after the conversion, the algorithmic steps are very similar, except that the bookkeeping information for H_ℓ is only implicit when H_ℓ is atomic. A similar situation obtains when H_ℓ has $D_{\overline{W}} = \emptyset$ either before or after the conversion. In all cases, the total time needed for the complete conversion is $O(r)$.

6 The Operations

This section describes how to execute the operations *color*, *setcolor* and *choice* and how to carry out an almost robust iteration.

color

In order to execute a call *color*(i), we identify the container H that stores the ith color value. The substructures D_W, $D_{\overline{W}}$ and D_{C} mentioned in the following are those of H. Accessing D_W, we can determine whether the color j to be returned is weak in H and, if so, j itself. If j is strong and $|\overline{W}| \geq 2$, we can access $D_{\overline{W}}$ to learn the code of j in H, from which j itself can be recovered using D_{C}. If j is strong and $|\overline{W}| = 1$, j is the unique element of $\overline{W}$. Once j is known, it is returned. The execution of *color* takes constant time.

setcolor

Suppose that a call of *setcolor* changes the color of some $i \in \{1, \ldots, n\}$ from j to j'. If j' is weak in the relevant container H and the operation causes the number of occurrences of j' in H to exceed r, H is first converted from j'-weak to j'-strong, as described in the previous section. The rest of the operation is straightforward. If j is weak in H, the pair (i, j) is removed from the substructure D_W of H, and if j' is weak in H, (i, j') is inserted in D_W. Similarly, if j' is strong in H and $|\overline{W}| \geq 2$, the code of j' in H is obtained from the substructure D_{C} of H, after which it is easy to carry out the appropriate update of the sequence $(a'_0, \ldots, a'_{s-1})$ stored in the substructure $D_{\overline{W}}$. A call of *setcolor* needs $O(r)$ time if it carries out a conversion of a container and constant time if not.

choice

The execution of a call *choice*(j) continues the sweep over the choice buffer for the color j, if any, until an occurrence of j is found or the sweep is complete. In the former case the position found is returned. In the latter case, as far as possible, a container H is selected that either belongs to L_j or is j-right – if there is no such container, there are no occurrences of j, and 0 is returned. If H is j-weak, we distinguish between two cases: If H belongs to L_j, it is not j-free, and an occurrence of j in H can be found in constant time and returned. If H is j-right and therefore a j-slave, it is replaced by its corresponding j-master. Now H is strong, a new choice buffer for the color j is derived from H, and the procedure is restarted recursively.

Almost Robust Iteration

Consider an iteration over a given color class S_j. In preparation for the iteration, we step through the list L_j to determine $|L_j|$. Then, using $O(\log n)$ bits of additional space, we split L_j into two lists L'_j and L''_j, with L'_j consisting of precisely those containers H_ℓ in L_j for which $\ell \le \mu_j - |L_j|^2$. The time needed to do this is $O(|L_j| + 1)$. Since the indices ℓ of the containers in L''_j are distinct integers in the range $\{\mu_j - |L_j|^2 + 1, \ldots, \mu_j\}$ of size $|L_j|^2$, we can sort the containers in L''_j by their indices with 2-pass radix sorting in $O(|L_j| + 1)$ time. If K is chosen sufficiently large, the $O(|L_j| \log n)$ bits of additional space needed for this can be supplied by the extra space, discussed shortly before Lemma 4.1, of the containers H_ℓ with $\mu_j - |L_j|^2 < \ell \le \mu_j$. At this point L_j is updated to be the concatenation of L'_j with the sorted L''_j.

Now the iteration proper can begin. We move a *token* through L_j, always robustly enumerating the elements of S_j in the *current container* H, the container that holds the token. If H is j-strong, this is done by deriving the iteration buffer for j from H and sweeping it, as described in Section 4. If H is j-weak, it is done similarly, by always remembering the last occurrence enumerated and using the fourth operation listed for D_W to enumerate the next occurrence. We will speak of a sweep also in this case. If a container H drops out of L_j because it becomes j-free while it holds the token, the token is first passed on to the successor of H in L_j, if any.

If a container H is j-strong or not j-free and H is to be inserted in L_j because it becomes j-left, then H is inserted at the end of L_j, i.e., so that it will still be swept. If a container is to be inserted in L_j because it stops being j-free, however, it is inserted at the beginning of L_j, i.e., so that it will not be swept.

Suppose that the iteration starts with $\mu_j = \widehat{\mu_j}$. If μ_j decreases all the way to $\widehat{\mu_j} - |L_j|^2$ (i.e., to the "border" between L'_j and L''_j) before all containers in L'_j have been fully swept, the enumeration is suspended and the containers in L_j whose sweep has not yet begun are sorted. Although it is not strictly necessary, this can happen "in-place" with Bubblesort in $O(|L_j|^2)$ time. Then the enumeration is resumed. When the iteration reaches the end of L_j, it proceeds to sweep the containers $H_{\overline{\mu_j}+1}, \ldots, H_N$ in that order, where $\overline{\mu_j}$ is the value of μ_j when the iteration over L_j finishes.

7 Analysis of Correctness and Execution Times

The correctness of the implementation of *color*, *setcolor* and *choice* is easy to see or has already been argued. Consider therefore an iteration over a color class S_j. Every element i that belongs to S_j throughout the iteration is stored in a container H that belongs to L_j or is j-right at the beginning of the iteration, and it is stored in H throughout any sweep over H. During the iteration H can drop out of L_j only by becoming j-right, and if H stops being j-right it is inserted at the end of L_j, where it will still be swept. Thus i is enumerated.

Let t_0 be the point in time when the iteration over L'_j ends or when what remains of L_j is sorted, whatever happens first. Until t_0 only containers in L'_j are iterated over, and any containers that enter L_j during this period are inserted at the beginning of L_j, where they will not be iterated over. Thus no element is enumerated more than once before t_0. After t_0 the enumeration happens strictly in increasing order, so again no element is enumerated more than once. Thus an element is enumerated at most once before t_0 and at most once after t_0. It is easy to see that no element is ever enumerated when it does not belong to S_j. It follows that the iteration is almost robust, as claimed.

The basic idea of the amortized timing analysis is simple: Disregarding iteration, all operations that involve only weak colors take constant time. Consider a point in time at which a container H is converted from j-weak to j-strong for some color j. At that time H contains more than r elements of color j. Because of this, before a conversion of H from j-weak to j-strong can happen again, a buffer for the color j must be derived from H, and the data structure must operate on each of the more than r elements of color j in H by enumerating the element, returning it (*choice*) or changing its color. We can "charge" all of the following to these more than r operations: The derivation of up to two buffers for color j from H, the sweep over the buffers, exclusive of the time spent reporting occurrences of j found there, and a possible conversion of H from j-strong to j-weak and back.

A problematic issue with the argument in the previous paragraph is that operations on elements in a buffer are called upon to pay for the derivation of the buffer, which happened earlier. For every color j other than 0 there is no problem here, since more than r operations – which can pay for the cost – must change the colors of elements in H to j before j can become strong in H for the first time. This argument does not apply to the color 0 because all elements have color 0 initially. However, using the structure D_W of H, a suitably represented first buffer for the color 0 can be derived from H in a time that is at most proportional to a constant plus the number of color changes executed on H since the initialization, so these operations can be charged with the cost.

Certain costs of an iteration are not covered by the analysis above (and, indeed, an iteration may temporarily "go into dept", which is why we cannot support incomplete iteration efficiently). First, there is the cost associated with sorting. The first sorting of part of L_j happens in $O(|L_j|+1)$ time, which is acceptable because every container that does not drop out of L_j before it receives the iteration token eventually contributes at least one occurrence of j to be enumerated or – if it turns out to be j-free – is converted from j-strong to j-weak, the cost of which was considered above. The second sorting of part of L_j is carried out, in $O(|L_j|^2)$ time, only after at least $|L_j|^2$ operations on elements of color j have been executed during the iteration, and the cost of the sorting can be charged to these operations. Second, there is the cost associated with sweeping j-slaves and j-weak former j-slaves that turn out to be j-free (so that the cost of the sweep cannot be charged to the occurrences found). This cost, however, can be charged to the sweeping of the corresponding masters or to the color changes that created or destroyed the matching links to these masters.

Theorem 4 of [8] furnishes a variant of the data structure of Lemma 4.1 that initializes all array elements to zero and can itself be initialized in constant time. Storing the global book-keeping information such as $\mu_0, \ldots, \mu_{c-1}$ in another instance of this data structure and interpreting the all-zero initial values appropriately allows the choice dictionary developed here to be initialized in constant time. We omit the details.

▶ **Theorem 7.1.** *There is a choice dictionary that, for arbitrary given positive integers n and c with $c = O((\log n)^{1/2}(\log\log n)^{-3/2})$, can be initialized with parameters n and c in constant time and subsequently occupies $n\log_2 c + O((\log n)^2)$ bits and executes color, setcolor and choice in constant amortized time and complete almost robust iteration (an element may be enumerated a second time) in constant amortized time per element enumerated.*

References

1 D. Angluin and L. G. Valiant. Fast Probabilistic Algorithms for Hamiltonian Circuits and Matchings. *J. Comput. Syst. Sci.*, 18(2):155–193, 1979. doi:10.1016/0022-0000(79)90045-X.

2 Niranka Banerjee, Sankardeep Chakraborty, and Venkatesh Raman. Improved Space Efficient Algorithms for BFS, DFS and Applications. In *Proc. 22nd International Conference on Computing and Combinatorics (COCOON 2016)*, volume 9797 of *LNCS*, pages 119–130. Springer, 2016. doi:10.1007/978-3-319-42634-1_10.

3 Preston Briggs and Linda Torczon. An Efficient Representation for Sparse Sets. *ACM Lett. Program. Lang. Syst.*, 2(1-4):59–69, 1993. doi:10.1145/176454.176484.

4 Yevgeniy Dodis, Mihai Pătraşcu, and Mikkel Thorup. Changing Base Without Losing Space. In *Proc. 42nd ACM Symposium on Theory of Computing (STOC 2010)*, pages 593–602. ACM, 2010. doi:10.1145/1806689.1806770.

5 Amr Elmasry, Torben Hagerup, and Frank Kammer. Space-Efficient Basic Graph Algorithms. In *Proc. 32nd International Symposium on Theoretical Aspects of Computer Science (STACS 2015)*, volume 30 of *LIPIcs*, pages 288–301. Schloss Dagstuhl – Leibniz-Zentrum für Informatik, 2015. doi:10.4230/LIPIcs.STACS.2015.288.

6 Torben Hagerup. Sorting and Searching on the Word RAM. In *Proc. 15th Annual Symposium on Theoretical Aspects of Computer Science (STACS 1998)*, volume 1373 of *LNCS*, pages 366–398. Springer, 1998. doi:10.1007/BFb0028575.

7 Torben Hagerup. Small Uncolored and Colored Choice Dictionaries. *Computing Research Repository (CoRR)*, abs/1809.07661 [cs.DS], 2018. arXiv:1809.07661.

8 Torben Hagerup. Highly Succinct Dynamic Data Structures. In Leszek Antoni Gasieniec, Jesper Jansson, and Christos Levcopoulos, editors, *Fundamentals of Computation Theory - 22nd International Symposium, FCT 2019, Copenhagen, Denmark, August 12-14, 2019, Proceedings*, volume 11651 of *Lecture Notes in Computer Science*, pages 29–45. Springer, 2019. doi:10.1007/978-3-030-25027-0_3.

9 Torben Hagerup. Fast Breadth-First Search in Still Less Space. In *Proc. 45th International Workhop on Graph-Theoretic Concepts in Computer Science (WG 2019)*, LNCS. Springer, 2019, to appear.

10 Torben Hagerup and Frank Kammer. Succinct Choice Dictionaries. *Computing Research Repository (CoRR)*, abs/1604.06058 [cs.DS], 2016. arXiv:1604.06058.

11 Torben Hagerup, Frank Kammer, and Moritz Laudahn. Space-Efficient Euler Partition and Bipartite Edge Coloring. *Theor. Comput. Sci.*, 754:16–34, 2019. doi:10.1016/j.tcs.2018.01.008.

12 Frank Kammer, Dieter Kratsch, and Moritz Laudahn. Space-Efficient Biconnected Components and Recognition of Outerplanar Graphs. *Algorithmica*, 81(3):1180–1204, 2019. doi:10.1007/s00453-018-0464-z.

13 Frank Kammer and Andrej Sajenko. Simple 2^f-Color Choice Dictionaries. In *Proc. 29th International Symposium on Algorithms and Computation (ISAAC 2018)*, volume 123 of *LIPIcs*, pages 66:1–66:12. Schloss Dagstuhl – Leibniz-Zentrum für Informatik, 2018.

14 Takashi Katoh and Keisuke Goto. In-Place Initializable Arrays. *Computing Research Repository (CoRR)*, abs/1709.08900 [cs.DS], 2017. arXiv:1709.08900.