Simple 2^f-Color Choice Dictionaries

Frank Kammer
THM, University of Applied Sciences Mittelhessen, Germany
frank.kammer@mni.thm.de

Andrej Sajenko[1]
THM, University of Applied Sciences Mittelhessen, Germany
andrej.sajenko@mni.thm.de

Abstract

A c-color choice dictionary of size $n \in \mathbb{N}$ is a fundamental data structure in the development of space-efficient algorithms that stores the colors of n elements and that supports operations to get and change the color of an element as well as an operation choice that returns an arbitrary element of that color. For an integer $f > 0$ and a constant $c = 2^f$, we present a word-RAM algorithm for a c-color choice dictionary of size n that supports all operations above in constant time and uses only $nf + 1$ bits, which is optimal if all operations have to run in $o(n/w)$ time where w is the word size.

In addition, we extend our choice dictionary by an operation union without using more space.

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

Keywords and phrases space efficient, succinct, word RAM

Digital Object Identifier 10.4230/LIPIcs.ISAAC.2018.66

Acknowledgements After acceptance of the paper, we find out that Torben Hagerup has independently developed a colored choice dictionaries [10]. His choice dictionary supports any number of colors, but is more complicated even if the number of colors is a power of two. We gratefully thank Torben Hagerup for several helpful comments while we prepared our final version.

1 Introduction

Data is already an important factor for many companies and is likely to become even more decisive in the future. The most successful business enterprises today are often those that have figured out, better than their competitors, how to collect and use data. Therefore, it is no surprise that the number of companies that store petabytes of data in warehouses grows quickly. E.g., Walmart stored over 40 petabytes of data already in 2017.

In order to provide better support for structured access, data is often stored in highly redundant forms. In a data warehouse, a data item is usually considered as a point in a multidimensional space. A typical operation is to intersect a data cloud with a hyperplane obtained by fixing a single attribute to a specific value – the so-called *slicing operation*. If space is of no concern, it is easy to support slicing by storing for each possible such hyperplane the data points that it contains. But space matters, and we need *dictionaries* – data structures for storing and retrieving information – with low redundancy. More generally, we need algorithms that are fast but also treat memory as a scarce resource. Therefore, such algorithms (see, e.g., [1, 2, 4, 5, 6, 7, 8, 11, 12, 13]) become increasingly relevant. An implementation of several space-efficient algorithms can be found on GitHub [14].

[1] Funded by the Deutsche Forschungsgemeinschaft (DFG, German Research Foundation) – 379157101.

In this paper, we focus on the improvement of a (c-color) choice dictionary, which is a fundamental data structure that is used in many of the algorithms listed above. A *choice dictionary* [3, 11] manages the membership for n elements. The main characteristic of the dictionary is that it can retrieve an arbitrary member if one exists. If the dictionary can manage more than two states (member or not), it is called a *c-color choice dictionary*.

▶ **Definition 1.** A c-color choice dictionary is a data type that can be initialized with an arbitrary integer $n \in \mathbb{N}$. Subsequently, it stores for each element $e \in U = \{1, \dots, n\}$ a color $q \in Q = [0, c-1]$, that is initially $q = 0$ and supports the following standard operations:

$\text{SETCOLOR}_q(e)$	Sets the color of element e to q.
$\text{COLOR}(e)$	Returns the color $q \in Q$ of element e.
CHOICE_q	Returns an (arbitrary) element of U that has the color q.

Another useful feature is an iterator that allows iterating over members of the dictionary and that can easily be used to support the slicing operation.

ITERATOR.INIT_q	Returns an iterator for an iteration over the q-colored elements.
$\text{ITERATOR.HASNEXT}_q$	Checks if the iterator can return a next element colored with color q.
ITERATOR.NEXT_q	Returns the next element of the iterator over the q-colored elements.

If we talk about an ITERATOR_q operation, then we mean the three operations ITERATOR.INIT_q, $\text{ITERATOR.HASNEXT}_q$ and ITERATOR.NEXT_q. The iterator needs $\Theta(\log n)$ bits. We call n the *size* of the choice dictionary. We assume that n is given to the data structure with each call of an operation. In this paper, we extend our choice dictionary by another operation $\text{UNION}_{q,q'}$ that recolors all q and q'-colored elements with one color.

Our model of computation is the word RAM, where we assume to have the standard operations to read, write, and modify a word of size $w = \Omega(\max\{\log n, \log c\})$ bits in constant time. The first c-color choice dictionary was defined by Hagerup and Kammer [11, Theorem 7.6]. Their choice dictionary supports initialization in constant time, all other operations in $O(t)$ time for $t \in \mathbb{N}$ and requires $n + O(n(\frac{t}{w})^t + \log n)$ bits of memory. For the special case of a 2-color choice dictionary, Hagerup [9] presented a choice dictionary that runs with $n + 1$ bits of memory, (i.e., only one bit of redundancy) and that supports COLOR, SETCOLOR_q, CHOICE_1, and ITERATOR_1 in constant time. He also showed that $n + 1$ bits are necessary unless the operations are allowed to run in $\Omega(n/w)$ time. He raised the open question how to support CHOICE_0 or ITERATE_0 and how to support several colors.

We answer the questions by introducing a c-color choice dictionary with c being a power of 2 that also uses only $n + 1$ bits of memory, and additionally supports CHOICE and ITERATOR for all colors in $O(c^3 \log c)$ time, which is constant for many applications since they use only a constant number of colors, i.e., $c = O(1)$. Since CHOICE_q can be easily implemented by using ITERATOR_q, we describe only the realization of the ITERATOR operation. We extend our choice dictionary with a union operation that, given two colors q and q', recolors all elements of one color to the other such that all operations run in amortized $O(1)$ time for $c = O(1)$.

The outline of the paper is as follows. In the next section, we describe known techniques and sketch their usage in the paper. In Section 3, we describe an extension of Hagerup's 2-color choice dictionary [9] to support CHOICE and ITERATOR for both colors. Afterwards, we extend some word RAM tricks for parallel computations within one word described by Hagerup and Kammer [11] for our usage with several colors. In Section 5, we generalize our choice dictionary to 2^f colors. We finally extend our choice dictionary by a UNION operation.

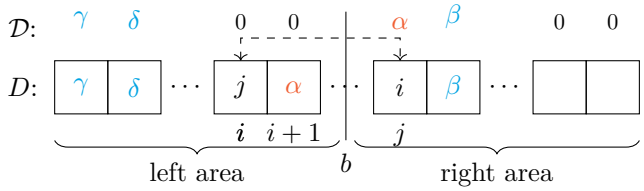

Figure 1 $\mathcal{D}$ shows the external view of the array and D represent the internal data structure. The variables i and j represent block numbers, α, β, γ and δ represent user defined words. The barrier b separates the array of blocks into a left and right area.

2 Previous Techniques

In this section we summarize the ideas introduced by Katoh and Goto [15] and Hagerup [9] to implement our c-color choice dictionary.

2.1 Initializable Array

Katoh and Goto use a two level approach to support constant time initialization of an array consisting of n bits. They divided the array into $O(n/w)$ *blocks* of $O(w)$ bits each, number the blocks from left to right with $1, 2, 3, \ldots$, and distinguish between two block states. A block can be either *initialized* (with user defined values) or *uninitialized* (containing arbitrary values). The initialization of one block is done in $O(1)$ time by setting its binary representation to zero. To implement the initialized array of n bits they determine if a block is initialized or uninitialized as follows – also sketched in Figure 1.

Partition the blocks using a *barrier* into a *left area* and a *right area*. All blocks in the left area are either initialized or have a so-called *chain* with an initialized block of the right area. A *chain* between an uninitialized block of the left area and an initialized block of the right area is created by writing the block number of each other in their first word. Because the initialized block contains values, but its first word is used to create the chain, they *relocate* the first word by storing it inside the second word of the chained uninitialized block. All blocks in the right area are uninitialized if they have no chain with a block of the left area.

To read a word of the array, determine to which block the word belongs to. Then determine if the block is inside the left or right area. If it is in the left area and has no chain, it is initialized and can be read and returned without further computations. If it has a chain, the block is uninitialized and zero can be returned. A block in the right area is uninitialized if it has no chain, so in this case a zero can be returned directly. If it has a chain, then the second word of the block can be read and returned directly, but for reading the first word, the chain must be followed to the block inside the left area and its second word must be returned.

Initially, the barrier is set before the first block and thus, the left area is empty and the array consists of only uninitialized blocks. If a block B of the left area is written, we must take care that no *unintended chains* are built, i.e., if the value in the first word points to an unchained block in the right area, then initialize that block with zeros. If a block B of the right area is written, it must be chained with an uninitialized block of the left area. Since we use the chain and unchain operations in the next sections, we describe them in detail.

The CHAIN operation takes a block B of the left area and a block B' of the right area, relocates the first word of B' into the last word of B and writes the block number of B' in the first word of B and the block number of B in the first word of B'. The UNCHAIN

operation takes a block B and returns an unchained block that needs to be chained. If B is not chained, initialize B with zeros and return it. If B is chained and in the left area, follow the chain pointer to a block B' of the right area, relocate the second word of B back to the first word of B' and initialize the block B with zeros. Finally, return B' to the caller. If B is in right area and B' chained to B, call this operation again and return $\textsc{unchain}(B')$.

We say that a block B is *connected to the left area* exactly if it is either inside the left area and unchained or inside the right area, but chained. Otherwise, it is called *disconnected*.

The $\textsc{connect}(B)$ operation works as follows. We assume that B is not connected. Increase the barrier by one. This increases the left area and moves a block B^* from the right area into the left area. We differ two cases. In a first case B is in the right area before the increase of the barrier. If $B = B^*$, then B is now connected, initialize all words of B with zeros and we are done. Otherwise, B^* is a candidate to create a chain with B. However, it can be already chained. No matter if it is chained or not, call $B' := \textsc{unchain}(B^*)$; note that the operation always returns an unchained block, possibly B^*. Chain it with B by calling $\textsc{chain}(B', B)$. In a second case B is in the left area. Then by definition it must have a chain with a block B'. To connect the block to the left area, the chain must be removed from B, but B' still requires a chain. Thus, call $B' = \textsc{unchain}(B)$, $B'' := \textsc{unchain}(B^*)$ and $\textsc{chain}(B'', B')$.

Writing blocks will cause blocks to connect to the left area and the left area to expand to the end of the array until every block belongs to it. Hagerup [9] introduced a technique how to disconnect blocks, i.e, moving the barrier from right to left such that blocks can become uninitialized again. This technique was also shown in the second version of [15].

The $\textsc{disconnect}(B)$ operation works exactly the other way around. We assume that B is connected. Decrease the barrier by one. This decreases the left area and moves a block B^* from the left area into the right area. If B is in the right area, call $B' := \textsc{unchain}(B)$, $B'' := \textsc{unchain}(B^*)$ and $\textsc{chain}(B', B'')$. If B is in the left area, then it needs a chain to be disconnected, call $B' = \textsc{unchain}(B^*)$ and $\textsc{chain}(B, B')$.

To use the approach above we need to store the barrier. That requires $O(w)$ extra bits of memory. To reduce the extra space needed to only 1 bit Katoh and Goto store the barrier inside the array as long not all blocks are initialized. If the array is fully initialized, i.e., all blocks are initialized, there is no need to chain blocks and the array is a normal array that can be read directly. One possible way to store the barrier in the array is to increase the block size to at least 3 words. Then the data of two chained blocks can be relocated such that the last word of every block inside the right area is always unused. (Either a block of the right area is uninitialized, i.e., all its word are unused, or is chained and therefore has one unused word left.) Since the left area expand to the right until the array is fully initialized, store the barrier inside the last unused word. Now an extra bit is used and set to 1 if the array is fully initialized and is a normal array and set to 0 if the array still has a right area of a barrier and therefore has to operate with blocks and chains.

2.2 Choice Dictionary with $\textsc{choice}_1$ and $\textsc{iterate}_1$

Hagerup implemented $\textsc{choice}_1$ and $\textsc{iterate}_1$ on an input consisting of n bits by using the techniques of Katoh and Goto. $\textsc{Choice}_1$ runs on $O(w)$ bits in constant time using word RAM operations. Let us call a block *non-empty* if it has at least one element (one bit) that is 1. Hagerup showed that all elements of uninitialized blocks of Katoh and Goto's initialized array can be seen as zero bits and also that by uninitializing blocks that contain only zeros, the left area contains only non-empty blocks or have a chain with a non-empty block. To support $\textsc{choice}_1$ one can pick an arbitrary block connected to the left area and to support $\textsc{iterate}_1$

one can move from the first block until the barrier and output the elements of the blocks connected to the left area. To output the elements of a block, again word RAM tricks can be used. Very recently, Hagerup [10] published a c-color choice dictionary for all $c \in I\!N$.

3 2-Color Choice Dictionary with CHOICE$_0$ and ITERATE$_0$

Hagerup supports CHOICE$_1$ and ITERATE$_1$ by managing the left area such that it only contains blocks that either have the color 1 or have a chain to a block with color 1. To support CHOICE and ITERATE on several colors the general idea is to manage such an area for each color, including color 0. In this section we assume to have only two colors $\{0, 1\}$, but since we want to support several colors later, we first give some definitions for an arbitrary number of colors before focusing on two colors.

We call a block that contains at least an element of color q a *q-block*, a block that has no such element a *q-free* block, a block that contains only the color q a *q-full* block and a block that contains all colors a *full* block. Moreover, we define two blocks as *q-chained* if their qth word contains the block number of each other and both blocks are separated by the barrier of color q. Keep in mind for the following algorithms that, if two blocks have a q-chain, the block in the left area of color q must be a q-free block. Moreover, each q-block is connected to the left area of color q, and each q-free block is disconnected from this area.

To support several areas we change the data structure as follows: We store a barrier b_q for each color $q \in \{0, \ldots, c-1\}$ and initially let them point before the first block of the array, except for the barrier of color 0. The barrier of color 0 points after the last block of the array. The reason for this is to support constant-time initialization even if we have an uninitialized memory. We define that a block B is uninitialized and is therefore defined as a block containing only the color 0 if B fulfills the following condition. B is in the right area for all colors $q \in \{1, \ldots, c-1\}$ and in the left area for color 0 (i.e., $\max\{b_1, \ldots, b_{c-1}\} < B \leq b_0$) as well as B has no q-chain for any color q. With this definition, initially all blocks are uninitialized and thus are 0-full blocks.

For the rest of the section, we focus on two colors. We assume now to have a block size of 3 words. Based on the ideas of Section 2.1, we next describe our invariant used to store the data in the blocks, which is also sketched in Figure 2. A block that has no chains simply stores its data unchanged – possibly, it is not initialized. A chained block uses its qth word ($q \in \{0, 1\}$) for a chain pointer of color q. Whenever a word of a chained block B in the right area is used for any q-chain to a block B' of the left area, then the user-data originally stored in the qth word of B is stored in the last word of the block B'. Assume now that the q-chain points to a block of the right area. Then we know from the chain that B is a $(1-q)$-full block, and we do not need to store any user-data of the block.

It remains to show that no block has two chains to the right area since, otherwise, both chains wants to store information in the last word of the block. Since we have only two colors, a block with a q-chain to the right must be a $(1-q)$-full block. Since no block can be 0-full and 1-full simultaneous, no block can have two chains to the right.

Iterating over elements of a color q can be done by iterating over the left area of color q and by determining the block type of each block B. Either B is a q-block or it is a q-free block chained with a q-block. Therefore, by using word RAM operations, we can output all elements of a color q in linear time.

We now describe how to change colors of the dictionary such that the properties described above are maintained. If we change the color of an element of a block B to a color q, we have to determine the block type first and then check if changing the color leads to a change of

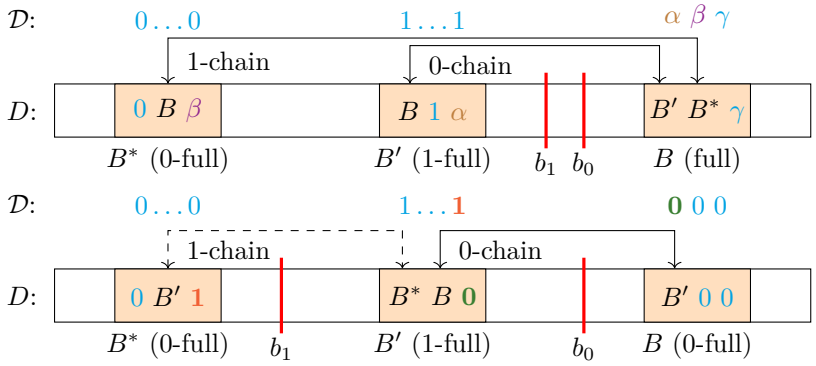

Figure 2 This figure shows the different situations of the internal data structure and external view that can occur with two barriers. The entries B, B', and B^* in the array D are block numbers.

the block type. Writing into a block changes its type only if we overwrite the last appearance of a color q' or introduce a new color q.

Similar to Section 2 we use the operations CONNECT and DISCONNECT to change the block type, but redefine it that it works on a specific color q: CHAIN and UNCHAIN consider chains and barrier (thus, areas and block types) with respect to color q. Whenever a color q is written to a q-free block B, we connect the block B to the left area of q. If the last appearance of a color q' has been overwritten after writing a color q, we disconnect the block from the left area of color q'.

Note that the chains and barriers of each color can be defined independently, and they do not interfere with the chains and barriers of other colors. Moreover, a block can change its type only if a color was introduced to it or has disappeared from it. After correcting the block type, write the color by modifying one word according to the new block type.

4 Word RAM Tricks

As long as we want to support only two colors, each element can be stored with only one bit. If the elements can have $c = 2^f$ colors for some integer $f > 1$, we have to use f bits to store the color, which we combine into a *field*. Thus, a word can store w/f colors, each in one field, and f is the size of the field. For simplicity, we assume that w is a multiple of f. If this is not the case, we may have to split the f bits of a color over two words, which does not change the asymptotic running time.

In the next section we want to modify several elements a_i in parallel that are located in fields part of one word in constant time. We next show some operations to realize the modifications.

The following lemma computes a bit mask consisting of 1 in the fields that have a value smaller than or equal to k. Let m and f be given integers with $1 \le m, f < 2^w$ and suppose that a sequence $A = (a_1, \ldots, a_m)$ with $a_i \in \{0, \ldots, 2^f - 1\}$ for all $i \in \{1, \ldots, m\}$ is given in form of the (mf)-bit binary representation of the integer $x = \sum_{i=0}^{m-1} 2^{if} a_{i+1}$. Then the following holds:

▶ **Lemma 2** ([11, Lemma 3.2c]). *Given a parameter $k \in \{0, \ldots, 2^f - 1\}$ in addition to x we can compute the integer $z = \sum_{i=0}^{m-1} 2^{if} b_{i+1}$ in $O(1 + mf/w)$ time, where $b_i = 1$ if $k \ge a_i$ and $b_i = 0$ otherwise for $i = 1, \ldots, m$.*

We next use the lemma above to recolor all elements in a word of one color into another color quickly.

▶ **Lemma 3.** *Given two colors q and q' in addition to x, we can compute the integer $z = \sum_{i=0}^{m-1} 2^{if} b_{i+1}$, where $b_i = a_i$ if $a_i \neq q$ and $b_i = q'$ otherwise for $i = 1, \ldots, m$.*

Proof. Apply the previous lemma twice on x, first with parameter $k = q$, then with k=$q - 1$, and subtract the second result from the first. We so get a vector v where the fields have a 1 exactly if the corresponding field in x has a q. If $q' < q$, then multiply v with $q - q'$ and subtract the result from x. Otherwise, $q' > q$ and multiply v with $q' - q$ and add the result to x. ◀

By applying the last lemma to all words in a block, we also get the following.

▶ **Corollary 4.** *Given the binary representation of a block consisting of the words $x_1, \ldots, x_j$ ($j \in \mathbb{N}$) and two colors q and q', we can recolor all q-colored elements in the block with color q' in $O(j)$ time.*

In the next section, a block has to store pointers for chains even if only one color is missing in the block. This means that we need a word to store the pointer, but the elements of every word in the block can have several colors. As already described in [11], the idea is to "pack" the information in the words of the block, which is possible since a color is missing. We again start with an auxiliary lemma.

▶ **Lemma 5** ([11, Part of the proof of Lemma 7.1])**.** *If all elements in x are different to a color q, then we can pack c subsequent elements into $cf - 1$ bits (instead of cf bits). We so get a* packed word*, i.e., a word where every (cf)th bit is not used to store the colors of the elements and therefore can be used to store other information. Both transformations (*pack *and* unpack*) run in $O(c)$ time, but the unpack operation needs to know the color q.*

We now show how to store extra words within a block if one color is missing in the block. In the next section, the words are used to store pointers for chains.

▶ **Lemma 6.** *Let B be a block consisting of $kc \log_2 c$ words and whose elements have a color in $\{1, \ldots, c\}$. If all elements in the block have a color different to $q \in \{1, \ldots, c\}$, then one can pack B in $kc^2 \log_2 c$ time such that we can additionally store kw bits within the block. We combine the kw bits to k* extra words*. In a packed block, $c \log_2 c$ time suffices to read and write an extra word, and given the color q, to unpack a word of the block. Thus, we can unpack the whole block in $kc^2 \log_2 c$ time.*

Proof. To pack B, run the algorithm from Lemma 5 on each word of the block. Note that the field size is $f = \log_2 c$. We so get kw bits that are not used to store the colors of the elements. Similarly, if q is given, we can unpack the words.

If B is packed, every cf bit is free. To read and write a word x that is stored within the free bits, we first need to compute a vector v of $w/(cf)$ fields of size cf with a 1 in all fields. We get v by using Lemma 2 with parameter $k = 2^{cf} - 1$ since all elements are $\leq k$.

To read an extra word, first run bitwise-and operation with v to zero the bits between the free bits. Then combine the free bits part of several words with bitwise-or operations and bit-shifts. We so combine the free bits of several words into one word, which then can be returned as an extra word. See also Figure 3.

To write an extra word, we first apply v suitable shifted on the extra word to split the given word into cf words such that each field is either 0 or 1. After clearing the free bits in the block using again the bitwise-and operation with v, we can use the bitwise-or to distribute the bits of the extra word to $c \log_2 c$ words. ◀

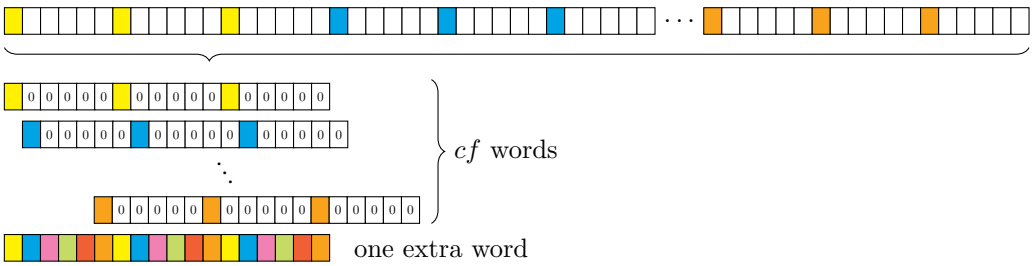

Figure 3 The construction of one extra word from bits distributed over $c \log_2 c$ words.

5 c-Color Choice Dictionary with CHOICE$_q$ and ITERATE$_q$

In Section 3, we focused on two colors. Now we increase the number of colors to $c = 2^f$, for an integer $f > 1$. We keep the concept of relocating words to create space for pointers. Again we exploit the fact that a block of the left area needs a chain if the block misses a color $q \in \{0, \ldots, c-1\}$. However, we need another idea to create space to store chain pointers because we can not determine the original content of a block by knowing the missing color. (With more than two colors, a q-free block is not automatically q'-full.)

The idea is to *pack* a block that misses a color by using Lemma 6 such that a q-free block has extra space to store information. With a block size of $(2c+1)c \log_2 c$ words, a packed block contains all its color information and has $2c+1$ extra words to store extra information. For the time being, we assume in the following that a packed block contains the $2c+1$ extra words at the beginning and then its packed color information. We use the first c words to store up to c chain pointers, the next c words to store relocated words of a chained full block, and the last extra word to store the missed color that was used to pack the block. The rest of the block contains the packed color information.

We are not able to pack a full block since it has no redundancy. A full block located in the left area of all colors does not need to be chained. However, a full block B in the right area of a color q requires a chain with a block B' that misses the color q. Thus, B' is packed and has $2c+1$ extra words. We use the qth word of B' to store a chain pointer to B, relocate the qth word of B into the $(c+q)$th word of B' and store the chain pointer to B' in the qth word of B. In contrast to the previous section, we only relocate words of a full block of the right area that requires a q-chain for some color q. If a block $\widetilde{B}$ is a q-block in the right area of a color q and misses some other color $q' \neq q$, $\widetilde{B}$ is packed and can store all its colors and all its pointers.

To read a block we need to check if the block is packed (i.e., not full) and needs to be unpacked first. We know that a block B is full exactly if, for each color q, either block B is in the left area of the color q and has no q-chain, or in the right area of color q for that B has a q-chain. To make this check we need to read chain pointers first. For each color $q \in \{1, \ldots, c-1\}$, we neither know if (1) a block is packed nor if (2) the qth word of a block stores colors or (3) a q-chain pointer. To find it out we have to check all q-chains. This is possible, since in all three cases (1) – (3), the qth chain is stored in the qth word. Thus, we only have to check if the qth word points to a block that points with its qth word back. Then B is q-chained. After checking it for all colors, we know if we are in case (1), (2) or (3).

The rest of the read operation is simple. Either we can read the words in the block directly or we have to unpack a word of a block, i.e., we first read the $(2c+1)$th extra word of the block where the color that was used to pack the block is located and use it to unpack the word such that the colors can be read.

Before we focus on the write operation, we start to discuss two problems and how we deal with them. We first describe how to handle the uninitialized area between the barrier of color 0 and the maximum barrier of all other colors, and second, how to avoid unintended chains. The uninitialized area may contain arbitrary values and only chained blocks are initialized in this area. These arbitrary values can be read as a pointer to a block that stores color information and these color information can be interpreted as a back pointer. In this situation the two blocks can have a chain that is unintended, and thus their block type changes without intention. In general, an unintended chain may be created whenever a block of the left area has been written with color information at a position where we also store pointers for chains. The solution of the problem extends the ideas from Katoh and Goto. We destroy unintended chains by checking if the word, where a chain pointer for a color q may be stored, creates a chain with another block. If it does, we destroy such a chain by writing n as an invalid chain pointer into the qth word of the unintended chained block in the uninitialized area. We also store n inside every of the first c extra words of a packed block that are not used to create chains. Since we use block numbers to create chains, the value n can not point to a block and thus can never create a chain. From now on we ignore the problems with uninitialized blocks and unintended chains.

We next describe the color change of an element e in a block B to color q. Recall from Section 3 that whenever we introduce a color q to a q-free block we connect the block to the left area of color q and whenever a color q' disappears from a q'-block we disconnect the block from the left area of color q'.

To color e in B with q, first determine if B is located in the left area of color q, the block type of B, and if B is packed. Then, run the corresponding case.

- ***B* is in the left area:**
  - ***B* is a *q*-free block:** Unpack the block B. Read the color q' of element e, overwrite it with color q and connect B with the left area of color q. Moreover, if B misses the color q', disconnect B from the left area of color q'. We now differ two cases.
    (1) The block becomes a full block. Note that B, as a full block, can not have chains to the right area. For all $\widetilde{q}$-chains to the left, with $\widetilde{q} \in \{0, \ldots, c-1\}$, relocate the $\widetilde{q}$th word of the unpacked block B to the $(c+\widetilde{q})$th word of the $\widetilde{q}$-chained block.
    (2) B misses some color q^*, possibly $q^* = q'$. Pack the block using color q^*.
  - ***B* is a *q*-block, but not full:** Unpack the block B. Read the color q' of element e and overwrite it with color q. If B contains no more q'-color elements, disconnect B from the left area of color q'. Finally, use the color that was previously used to pack the block and pack it again.
  - ***B* is a full:** Thus, B is already unpacked. Read the color q' of element e and overwrite it with color q. If B misses no color, then we are done. Otherwise, let q' be the missing color. Disconnect B from the left area of color q'. We next want to pack B. Before, we have to relocate back B's first c words for all chains of B (all chains are to the left), i.e, for all $\widetilde{q}$-chains, with $\widetilde{q} \in \{0, \ldots, c-1\}$, move the $(c+\widetilde{q})$th word in the $\widetilde{q}$-chained block of B to $\widetilde{q}$th word in B. Then, pack B using color q'.
- ***B* is in the right area:**
  - ***B* is a *q*-free block:** Unpack block B. Read the color q' of element e and overwrite it with color q. Connect B to the left area of color q and if B misses the color q', disconnect B from the left area of color q'. We differ two cases.
    (1) B misses a color q^*, possibly $q^* = q'$. Pack the block using color q^*.
    (2) Otherwise, B becomes a full block. Relocate B's first c words into the chained blocks, i.e, for all $\widetilde{q}$-chains, with $\widetilde{q} \in \{0, \ldots, c-1\}$ the $\widetilde{q}$th word in B to the $(c+\widetilde{q})$th

word of the block $\widetilde{q}$-chained with B.

- ***B* is a *q*-block, but not full:** Unpack block B. Read the color q' of element e and overwrite it with color q. If color q' is not present in B anymore, disconnect it from the left area of color q', and use the color that was previously used to pack the block.
- ***B* is a full and unpacked:** Read the color q' of element e and overwrite it with color q. If B misses color q', we want to pack B again. Thus, copy the $(c+\widetilde{q})$th word of every $\widetilde{q}$-chained block $\widetilde{B}$ into the $\widetilde{q}$th word of B, for $\widetilde{q} \in \{0, \ldots, c-1\}$. Pack B using color q' and disconnect B from the left area of color q'.

Since we want to use Lemma 6, the extra words of a packed block are not a sequence of consecutive bits. They are distributed inside the block over every $(c \log_2 c)$th bit. Therefore, we can not store the pointers in a full block simply at the beginning and relocate one continuous word. Instead, we store our pointers in the same distributed way and the relocation saves the distributed bits.

To relocate the bits, we can simply use the read operation of Lemma 6 to get the distributed bits in compact words, which then can be relocated. Analogously, to read and write the pointer of a chain, we can use the read and write operation, respectively, of the lemma. Note that the read and write operation also works even if the words are not packed.

We now describe how we store the barriers in the dictionary as long as not every block contains all colors such that the dictionary requires only 1 extra bit. We increase the block size to $(3c+1)c \log_2 c$ words, and increase so the number of extra words in a block by c, which allows us to store the c barriers of all colors within one pair of chained blocks. Since all the barriers moves to the right, the rightmost block is always packed or has a chain to a packed block unless all blocks have all colors. Then, we do not require to store the barriers. The only information that we have to store is a one bit that is set to 1 exactly if every block contains all colors and the whole dictionary is a normal array that can be read by accessing the data directly.

The size n of our 2^f-color choice dictionary is necessary to decide if the barriers reached the end of the array or not. As long as we have chains, we can store the size n in the first word and move the original content from there into some extra word. However, if all blocks are full – i.e., we have maximal entropy – we need fn bits to store the colors. Thus, if we reintroduce the barriers again (a color disappears from a block), we need to know the size n from the user. So, we assume that the user provides the size whenever an operation is called.

▶ **Theorem 7.** *For integers $f \geq 1$ and $c = 2^f$, there is a c-color choice dictionary that occupies $nf + 1$ bits of memory and supports all standard operations in $O(c^3 \log c)$ time.*

6 Operation UNION

The UNION operation takes two colors q and q', iterates over all elements of one of these colors and recolors them with the other color. After recoloring the elements, it returns the color that was chosen. It is not user controlled which color is chosen as the new color.

To implement UNION(q, q') in amortized constant time we select the color that appears in fewer (or the same number of) blocks, say q. We know this by comparing the size of the left areas of q and q', i.e, by comparing the barriers of the two colors. If the extra bit of the choice dictionary is 1, the left areas are of equal size. Then, iterate over all blocks that are connected to the left area of q and recolor all q in q' in that block in constant time per word (Corollary 4).

▶ **Theorem 8.** *For integers $f \geq 1$ and $c = 2^f$, there is a c-color choice dictionary that occupies $nf + 1$ bits of memory and supports* SETCOLOR$_q$ *for $q \in \{0, \ldots, c-1\}$ in $O(c^4 \log c)$ amortized time, all remaining standard operations in $O(c^3 \log c)$ amortized time as well as a* UNION *operation in amortized constant time.*

Proof. To define a potential function, let k_q be number of blocks that have a q-colored element, and let $\sigma : \{0, \ldots, c-1\} \to \{0, \ldots, c-1\}$ be a permutation so that $k_{\sigma(0)} \geq \cdots \geq k_{\sigma(c-1)}$. We now take $\Phi = C \sum_{q=0}^{c-1} k_{\sigma(q)} q$ as the potential function for our amortized running time analysis where $C > 0$ is some integer defined below. In other words, each block with a q-colored element gives us a contribution of $C\sigma^{-1}(q)$ to our potential function. Note that a necessary change of σ can be always done in such a way that Φ does not change: Before $k_{\sigma(q)} \geq k_{\sigma(q')}$ becomes wrong due to an increase of $k_{\sigma(q')}$ or a decrease of $k_{\sigma(q)}$ for some colors $q, q' \in \{0, \ldots, c-1\}$ with $\sigma(q) = \sigma(q') + 1$, we have $k_{\sigma(q)} = k_{\sigma(q')}$. Consequently, we can interchange the values of $\sigma(q)$ and $\sigma(q')$ without a change of Φ.

It is easy to see that the operations COLOR and ITERATOR do not change Φ. Let us now consider a call of operation UNION on colors q and q'. Assume that $k_q \geq k_{q'}$ and thus $\sigma(q) < \sigma(q')$. UNION iterates through all blocks with color q' and recolors the q'-colored elements in the blocks. This can be done in $O(k_{q'} c^3 \log c) = Ck_{q'}$ total time by choosing $C = C' c^3 \log c$ for some constant $C' > 0$. Since $\sigma(q) < \sigma(q')$, Φ shrinks with each recoloring of all q'-colored elements in a block by $C(\sigma(q') - \sigma(q)) \geq C$ since $k_{q'}$ decreases by one and k_q increases by at most one. In total, Φ shrinks by at least $Ck_{q'}$ so that the amortized costs are 0.

Finally, note that the operation SETCOLOR$_q$ increases Φ by at most $C(c-1)$ so that its amortized running time is $O(Cc + c^3 \log c) = O(c^4 \log c)$. ◀

We finally show that our amortized analysis of the last proof is tight, i.e., if we want to support an amortized constant-time UNION operation, the amortized time to color an element increases by a factor $\Omega(c)$ to pay for the recoloring that are done by the UNION operations. We show this by constructing an instance where at least half of the elements are moved $c-2$ times, i.e, the UNION operation recolors the elements $c-2$ times. It suffices to consider only one fixed instance since we can easily scale the instance size by any factor z; simply replace each word below by z copies having the same colors.

Let us assume that every word has one ball in each color that occurs in the word, and let us assign the balls to $c-1$ columns of a table as follows. A ball having color i is part of column i for each $i \in \{1, \ldots, c-1\}$. Initially, we have only zero words. Those words have no balls at all. We next construct an instance by filling first column 1, then column 2, etc. and show by induction the following property: the fraction of balls in column $i \geq 1$ that are moved by $i-1$ steps is at least $1 - i/2c$. Moreover, let k_i be the number of balls in the ith column of our instance constructed.

The induction clearly holds for $i = 1$ by adding one ball to the 1st column (color one element with color 1) and we can take $k_1 = 1$. We next show how to obtain the property for the ith column ($i > 1$) and assume that the property holds for $i-1$. Add $k_{i-1} + 1$ new balls to the ith column (i.e., color one element in $k_{i-1} + 1$ many zero words with color i) – these balls are moved 0 steps in the table. Use induction and build column $i-1$ with k_{i-1} balls such that the induction property holds for that column, move the balls to the ith column (run a UNION operation on $i-1$ and i) and repeat the steps in this sentence $x-1$ times. We so get a fraction of $(x - 1/x)(1 - (i-1)/2c)$ balls that are moved $i-1$ steps, i.e., the balls have visited columns $1, \ldots, i$. Clearly, $k_i = xk_{i-1} + 1$. By choosing x large enough we get $(x - 1/x)(1 - (i-1)/2c) \geq (1 - i/2c)$ since $(1 - (i-1)/2c) \geq (1 - i/2c)$ and $(x - 1/x) \to 1$ for $x \to \infty$. This shows the property for column i.

References

1. Tetsuo Asano, Amr Elmasry, and Jyrki Katajainen. Priority Queues and Sorting for Read-Only Data. In *Proc. 10th International Conference on Theory and Applications of Models of Computation (TAMC 2013)*, volume 7876 of *LNCS*, pages 32–41. Springer, 2013. doi:10.1007/978-3-642-38236-9_4.
2. Tetsuo Asano, Taisuke Izumi, Masashi Kiyomi, Matsuo Konagaya, Hirotaka Ono, Yota Otachi, Pascal Schweitzer, Jun Tarui, and Ryuhei Uehara. Depth-First Search Using $O(n)$ Bits. In *Proc. 25th International Symposium on Algorithms and Computation (ISAAC 2014)*, volume 8889 of *LNCS*, pages 553–564. Springer, 2014. doi:10.1007/978-3-319-13075-0_44.
3. 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)*, volume 9797 of *LNCS*, pages 119–130. Springer, 2016. doi:10.1007/978-3-319-42634-1_10.
4. Sankardeep Chakraborty, Anish Mukherjee, Venkatesh Raman, and Srinivasa Rao Satti. A Framework for In-place Graph Algorithms. In *Proc. 26th Annual European Symposium on Algorithms (ESA 2018)*, volume 112 of *LIPIcs*, pages 13:1–13:16. Schloss Dagstuhl – Leibniz-Zentrum für Informatik, 2018.
5. Omar Darwish and Amr Elmasry. Optimal Time-Space Tradeoff for the 2D Convex-Hull Problem. In *Proc. 22nd Annual European Symposium on Algorithms (ESA 2014)*, volume 8737 of *LNCS*, pages 284–295. Springer, 2014. doi:10.1007/978-3-662-44777-2_24.
6. Samir Datta, Raghav Kulkarni, and Anish Mukherjee. Space-Efficient Approximation Scheme for Maximum Matching in Sparse Graphs. In *Proc. 41st International Symposium on Mathematical Foundations of Computer Science (MFCS 2016)*, volume 58 of *LIPIcs*, pages 28:1–28:12. Schloss Dagstuhl – Leibniz-Zentrum für Informatik, 2016. doi:10.4230/LIPIcs.MFCS.2016.28.
7. 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.
8. Amr Elmasry and Frank Kammer. Space-Efficient Plane-Sweep Algorithms. In *Proc. 27th International Symposium on Algorithms and Computation (ISAAC 2016)*, volume 64 of *LIPIcs*, pages 30:1–30:13. Schloss Dagstuhl – Leibniz-Zentrum für Informatik, 2016. doi:10.4230/LIPIcs.ISAAC.2016.30.
9. Torben Hagerup. An Optimal Choice Dictionary. *CoRR*, abs/1711.00808, 2017. arXiv:1711.00808.
10. Torben Hagerup. Small Uncolored and Colored Choice Dictionaries. *CoRR*, abs/1809.07661, 2018. arXiv:1809.07661.
11. Torben Hagerup and Frank Kammer. Succinct Choice Dictionaries. *CoRR*, abs/1604.06058, 2016. arXiv:1604.06058.
12. Torben Hagerup, Frank Kammer, and Moritz Laudahn. Space-efficient Euler partition and bipartite edge coloring. *Theor. Comput. Sci.*, 2018. doi:10.1016/j.tcs.2018.01.008.
13. Frank Kammer, Dieter Kratsch, and Moritz Laudahn. Space-Efficient Biconnected Components and Recognition of Outerplanar Graphs. *Algorithmica*, 2018. doi:10.1007/s00453-018-0464-z.
14. Frank Kammer and Andrej Sajenko. Space efficient (graph) algorithms. https://github.com/thm-mni-ii/sea, 2018.
15. Takashi Katoh and Keisuke Goto. In-Place Initializable Arrays. *CoRR*, abs/1709.08900, 2017. arXiv:1709.08900.