Extra Space during Initialization of Succinct Data Structures and Dynamical Initializable Arrays

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

Many succinct data structures on the word RAM require precomputed tables to start operating. Usually, the tables can be constructed in sublinear time. In this time, most of a data structure is not initialized, i.e., there is plenty of unused space allocated for the data structure. We present a general framework to store temporarily extra buffers between the user defined data so that the data can be processed immediately, stored first in the buffers, and then moved into the data structure after finishing the tables. As an application, we apply our framework to Dodis, Pătraşcu, and Thorup's data structure (STOC 2010) that emulates c-ary memory and to Farzan and Munro's succinct encoding of arbitrary graphs (TCS 2013). We also use our framework to present an in-place dynamical initializable array.

2012 ACM Subject Classification Theory of computation $\rightarrow$ Data structures design and analysis

Keywords and phrases space efficiency, succinct c-ary memory, dynamic graph representation

Digital Object Identifier 10.4230/LIPIcs.MFCS.2018.65

1 Introduction

Small mobile devices, embedded systems, and big data draw the attention to space- and time-efficient algorithms, e.g., for sorting [4, 23], geometry [1, 3, 10] or graph algorithms [2, 6, 7, 9, 12, 14, 15, 16]. Moreover, there has been also an increased interest in succinct (encoding) data structures [8, 11, 12, 20, 18, 24, 22].

On a word RAM, succinct data structures often require precomputed tables to start operating, e.g., Dodis, Pătraşcu, and Thorup's data structure [8]. It emulates c-ary memory, for an arbitrary $c \geq 2$, on standard binary memory almost without losing space. Before we can store any information in the data structure, suitable lookup tables have to be computed. Hagerup and Kammer [12, Theorem 6.5] showed a solution that allows us to store the incoming information in an extra buffer and read and write values immediately. However, their solution scrambles the data so that extra mapping tables have to be built and used forever to access the data, even after the lookup tables are built. Moreover, Farzan and Munro [11] described a succinct encoding of arbitrary graphs. To store a dense graph, an adjacency matrix is stored in a compact form by decomposing the matrix in tiny submatrices. Each submatrix is represented by an index and the mapping is done via a lookup table. Assume that the tables are not ready, but the information of the graph already arrives in a

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

stream. One then can use an extra buffer and store the small matrices in a non-compact way and operate on these matrices until the tables are ready. At the end, the non-compact matrices can be moved from the buffers to the data structure that stores the encoded graph.

Usually, the tables can be constructed in sublinear time. In this time, most of the data structure is not initialized, leaving plenty of allocated space unused. We present a general framework to store a temporarily extra buffer within the initial memory. Even if the initialized space changes, the content in the extra buffers do not change for the user. Intuitively, we describe a way how to use the uninitialized space to store a usual array with random access. In particular, if the buffer is not needed any more and thus is removed, our implementation "leaves" the data structure as if the extra buffer has never existed.

As another application of our framework, we show that in-place initializable arrays can be made dynamic and with every increase of the array size, the new space is always initialized. Our model of computation is the word RAM model with a word length w that allows us to access all input words in $O(1)$ time. Thus, to operate on a dynamic initializable array of maximum size $n_{\max}$ we require that the word size $w = \Omega(\log n_{\max})$.

To obtain a dynamic initializable array, we additionally assume that we can expand and shrink an already allocated memory. The memory can come from an operating system or can be user controlled, i.e., the user can shrink an initializable array temporarily, use the space for some other purpose, and can increase it again if it is needed later.

1.1 Previous Array Implementation

The folklore algorithm [5, 19], which uses two arrays with pointers pointing to each other and one array storing the data, uses $O(nw)$ bits of extra space. Navarro [21] showed an implementation that requires $n + o(n)$ bits of extra space. Recently, Hagerup and Kammer [13] showed that $\lceil (n(t/(2w))^t) \rceil$ extra bits for an arbitrary t suffice if one wants to support access to the array in $O(t)$ time. In particular, by choosing $t = O(\log n)$, the extra space is only one bit. Very recently, Katoh and Goto [17] showed that also constant access time is possible with one extra bit. All these array implementations are designed for static array sizes.

1.2 Our Contributions

First, we present our framework to extend a data structure by an extra buffer. We then apply the framework to two known algorithms [8, 11]. Finally, we make Katoh and Goto's in-place initializable arrays dynamic. For this purpose, we identify problems that happen by a simple increase of the memory used for the array and stepwise improve the implementation to make our array dynamic. Our best solution supports operations READ, WRITE, and INCREASE in constant time and an operation SHRINK in amortized constant time.

Why do we support the shrink operation only in amortized time? The rest of this paragraph is not a precise proof, but gives some intuition what the problems are if the goal is to support constant non-amortized time for initialization, writing and shrinking. To support constant-time initialization of an array we must have knowledge of the regions that are completely written by the user and these regions must store the information which words are written. If we now allow arbitrary shrink operations, then we do not have the space to keep all information. Instead, we must clean the information, but keep this information of the written words that still belong to the dynamic array. If we want to run the shrink operation in non-amortized constant time, the information must be sorted in such a way that the cleaning can be done by simply cutting away a last part of the information. This means the writing operation must take care that the information is stored "almost sorted". Since we can not sort a stream of elements in $O(1)$ time per element in general, it seems plausible that the writing operation takes $\omega(1)$ time.

The remainder of this paper is structured as follows. We begin with summarizing the important parts of Katoh and Goto's algorithm in Section 2. In Section 3 we present our framework to implement the extra buffer stored inside the unused space of a data structure without using extra space. In Section 4, we extend our framework to dynamic arrays. As an application, we show in Section 5.1 how to increase the array size in constant time. In Sections 5.2 and 5.3, we present the final implementation that also allows us to decrease the array size.

2 In-Place Initializable Array

Katoh and Goto [17] introduced the data structure below and gave an implementation that uses $nw + 1$ bits and supports all operations in constant time.

▶ **Definition 1.** An initializable array $\mathcal{D}$ is a data structure that stores $n \in I\!N$ elements of the universe $Z \subseteq I\!N$ and provides the following operations:

- READ(i) $(i \in I\!N)$: Returns the i-th element of $\mathcal{D}$.
- WRITE(i, x) $(i \in I\!N, x \in Z)$: Sets the i-th element of $\mathcal{D}$ to x.
- INIT(x) $(x \in Z)$: Sets all elements of $\mathcal{D}$ to initv $:= x$.

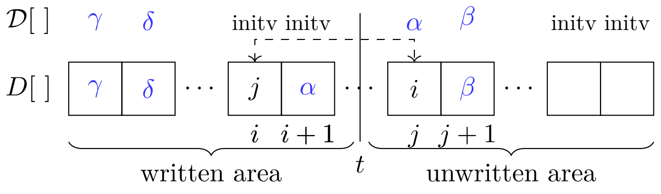
Let $D[0, \ldots, n]$ be an array storing $\mathcal{D}$. $D[0]$ is used only to store one bit to check if the array $\mathcal{D}$ is fully initialized. If $D[0] = 1$, $\mathcal{D}$ is a normal array, otherwise the following rules apply.

The idea is to split a standard array into blocks of $b = 2$ words and group the blocks into two areas: The blocks before some threshold $t \in I\!N$ are in the *written area*, the remaining in the *unwritten area*. Moreover, they call two blocks *chained* if the values of their first words point at each others position and if they belong to different areas. In the following, we denote by $d(B)$ the block chained with a block B. Note that $d(d(B)) = B$.

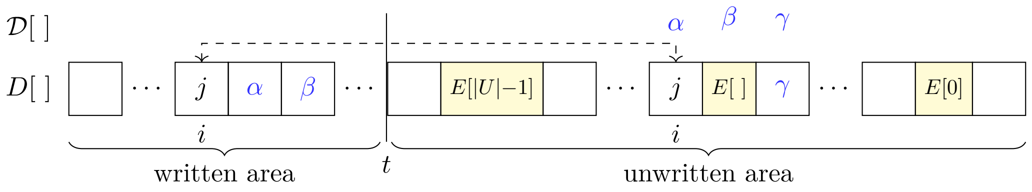
In our paper, we also use chains. Therefore, we shortly describe the meaning of a (un)chained block B in [17]. Compare the following description with Figure 1. If B is in the written area and chained, then B is used to store a value of another initialized block and is therefore defined as uninitialized. If B is in the unwritten area and chained, B contains user written values and the words of B are divided among B and $d(B)$. In this case $d(B)$ was not written by the user. In detail, the second word of B is stored in this word of B, but the first word is stored in the second word of $d(B)$ since the first word is used to store a pointer. If B is in the written area and unchained, B contains user written (or initial) values. If B is in the unwritten area and unchained, B has never been written by the user. Every time the user writes into a block for the first time, i.e., into a word of an (un)chained block in the (un)written area, the written area increases by moving the first block of the unwritten area into the written area and by possibly building or correcting chains.

3 Extra Space during Initialization of Succinct Data Structures

We now consider an arbitrary data structure $\mathcal{D}^*$. The only assumption that we make is that $\mathcal{D}^*$ accesses the memory via a data structure $\mathcal{D}$ realizing an n-word array. For the time being, assume $\mathcal{D}$ is the data structure as described in Section 2 with $D[0, \ldots, n]$ being an array storing $\mathcal{D}$. We define $|U|$ as the number of blocks of the unwritten area. As long as



■ **Figure 1** The different states of blocks inside the written and unwritten area. $\mathcal{D}$ represents the users view on the data with *initv* being the initial value, α, β, γ and δ as user written values. D represents the internal view with i and j as indices of D.



■ **Figure 2** The block size $b = 3$ allows us to move α and β to the chained block such that each block of the unwritten area has one unused word to store the extra data of E.

$\mathcal{D}$ is not fully initialized, i.e, not all array locations have been written since the last INIT operation, and only $m \in \mathbb{N}$ with $m < n$ words are written in $\mathcal{D}$, Theorem 2 shows that the unused storage allocated for $\mathcal{D}$ allows us to store information of an extra array E inside D by increasing the block size to $b > 2$ (Figure 2). The size of E depends on $|U|$ and thus on the current grade of initialization of $\mathcal{D}$. In contrast to $\mathcal{D}$, E is an uninitialized array. Clearly, we can build an initialized array on top of E.

The array size n is not always a multiple of the block size b. By decreasing n and handling less than b words of $\mathcal{D}$ separately, we assume in this paper that n is a multiple of b.

▶ **Theorem 2.** *We can store an extra array E of $(n/b - m)(b - 2) \leq |U|(b - 2)$ words inside the unused space of an initializable array $\mathcal{D}$ of size n, split into blocks of size b where m is the number of writing operations since the last* INIT *operation. E dynamically shrinks from the end whenever the number of user defined values m increases.*

Proof. Let us consider a block B inside the unwritten area. B and $d(B)$ together provide space for $2b$ words, but the chain between them exists only because the user wrote inside one block (block B, but not in $d(B)$), and thus we need to store b words of user data. Furthermore, B and $d(B)$ each needs one word to store a pointer that represents the chain. By storing as much user data as possible of B in $d(B)$, we have $b - 2$ unused words in B – say, words 2 to $b - 1$ are unused. Note that, if a block B is unchained, then it contains no user values at all and we have even b unused words. If the user wrote m different words in $\mathcal{D}$, then the threshold t of $\mathcal{D}$ (t is the number of blocks in the written area) is expanded at most m-times. The written area consists of at most m blocks of a total of n/b blocks. Consequently, the unwritten area has $(n/b - m) \leq |U|$ blocks with each having $(b - 2)$ unused words. If we start storing $E[0], E[1], \ldots$ in the unwritten area strictly behind t, the indices of E will shift every time the unwritten area shrinks. Therefore, we store E in reverse and E loses with every shrink the last $b - 2$ words, but get static indices. ◀

Note that values like the threshold t and the initial value can be easily stored at the beginning of E. However, we require the size of $\mathcal{D}$ to determine the beginning of E. To store the size we therefore move $D[1]$ also into E and store the size of the array in $D[1]$. During the usage of D we have to take this into account, but for simplicity we ignore this fact. If the array is fully initialized and thus $D[0] = 1$, we can not store the size anymore, but in this case we do not need to know it.

3.1 Application: c-ary Memory

To use Dodis et al.'s dictionary [8] on n elements, a lookup table Y with $O(\log n)$ entries consisting of $O(\log n)$ bits each must be constructed, which can be done in $O(\log n)$ time. Using tables of the same kind to store powers of c, we can assume that $c = \Omega(n)$ by combining consecutive elements of the input into one element.

Hagerup and Kammer extended Dodis et al.'s dictionary to support constant-time initialization by storing the $k = \Theta(\log n)$ elements that are added first to the dictionary in a trie D^{T} of constant depth $d \geq 4$ and out degree $n^{1/d}$ using $O(n^{\epsilon+1/d} + \log n \log c)$ bits for any $\epsilon > 0$. Interleaved with the first k operations on the dictionary, first the table Y is built and afterwards, the elements in D^{T} are moved to the dictionary.

Since D^{T} is required only temporarily, it is stored within the memory allocated for the dictionary – say in the last part of the memory. The problem that arises is that the last part of the memory can not be used as long as it is still used by D^{T}. This problem is solved by partitioning the memory allocated for the dictionary into sectors and using a complicated mapping function that scrambles the elements to avoid the usage of the last part of the memory. Unfortunately, even after computing the table Y and moving all elements from D^{T} over to the dictionary, the data is still scrambled and each access to the dictionary has to start evaluating the mapping function.

With Theorem 2 as an underlying data structure it is easily possible to implement the dictionary with constant initialization time. Use the extra array E to store temporarily the table D^{T}. Even if we use block size $b = 3$, E has enough space (at least $(n(\log c) - k \log n)/3 \geq (n-k)(\log n)/3$ bits at the end of the construction of Y) to store D^{T}. The mapping function becomes superfluous because the data is not scrambled by the usage of Theorem 2. Furthermore, working on a copied and slightly modified algorithm after the full initialization of $\mathcal{D}$, we can avoid checking the extra bit in $D[0]$.

3.2 Application: Succinct Encoding of Dense Graphs

Farzan and Munro [11] showed a succinct encoding of an $n \times n$-matrix that represents an arbitrary graph with n vertices and m edges. Knowing the number of edges they distinguish between five cases: An almost full case, where the matrix consists of almost only one entries, an extremely dense case, a dense case, a moderate case and an extremely sparse case. For each case, they present a succinct encoding that supports the query operations for adjacency and degree in constant time and iteration over neighbors of a vertex in constant time per neighbor.

We consider only the dense case where table lookup is used. Dense means that $\exists \delta > 0 : n^2/\log^{1-\delta} n \leq m \leq n^2(1 - 1/(\log^{1-\delta} n)$. As shown in [11], a representation in that case requires $\log \binom{n^2}{m} + O(n^2(\log^{1-\delta} n)) = \log \binom{n^2}{m} + o(\log \binom{n^2}{m})$ bits of memory. To encode the matrix of a graph Farzan and Munro first divide the matrix into small *submatrices* of size $\log^{1-\delta} n \times \log^{1-\delta} n$ for a constant $0 < \delta \leq 1$. For each row and column of these smaller submatrices they calculate a *summary bit* that is 1 if the row and column, respectively, of the submatrix contains at least one 1. The summary bits of a row and column, respectively, of the whole matrix are used to create a *summary vector*. On top of each summary vector they build a rank-select data structure [24] that supports queries on 0's and 1's in constant time. They also build a lookup table to map between possible submatrices and indices as well as to answer all queries of interest in the submatrix in $O(1)$ time. In a further step, they replace each submatrix by its index. By guaranteeing that the number of possible submatrices is $o(\log n)$, the construction of the lookup table can be done in $O(n)$ time. To simply guarantee this, we restrict δ to be larger than $1/2$. We generalize the result to dynamic graphs by replacing the rank-select data structures with choice dictionaries [12, Theorem 7.6] as follows. We assume that the graph has initially no edges and that there is a stream that consists of edge updates and query operations. With each edge update, the index of the submatrix with the edge changes. To realize the index transition we use a translation table that maps both, the current index of the submatrix and the edge update made, to the new index. We store for each summary vector an *edge counter*, i.e., the number of 1's that was used to calculate each summary bit of it. Whenever an edge is created between two nodes we set the corresponding bit of the row and column, respectively, inside the summary vector to 1 and increment the edge counter. If an edge is removed we decrement the edge counter and determine, using a lookup table, if the submatrix containing this edge has still 1's in the updated row and column, respectively. If it does not, we set the corresponding summary bit inside the summary vector to 0.

If the lookup tables for the submatrices are computed, we can easily answer queries to the current graph similar to [11] since we still use the summary vectors. We can ask for membership to answer adjacency queries. To iterate over the neighbors of a vertex (i.e., over the 1's in the summary vector), we can use the iterator function of the choice dictionary instead of using the select function of the rank-select data structure. The degree query is answered by returning the edge counter.

In the rest of this subsection, the goal is to extend the data structure such that it supports the queries without a delay for the construction of the lookup tables. The idea is to store a submatrix with a 1 entry in the usual, non-compact way in the extra buffer and to add a pointer from the submatrix to a place in the buffer where it is stored. To reduce the space used for the pointers, we group $\log^{\delta} n$ submatrices together to get a *group matrix* of size $\log^{1-\delta} n \times \log n$. Moreover, we use an array A in which we store a pointer of $O(\log n)$ bits for each group. With the first writing operation to one submatrix in a group, the group matrix is stored in the usual, non-compact way in an initialized array, which is stored in the extra buffer (Theorem 2). In the array A, we update the pointer for the group. In addition, we build a choice dictionary on top of each non-empty column and each non-empty row of the group matrix. With each choice dictionary, we also count the number of 1 entries.

Using the counts we can update the summary vectors and the non-compact submatrix allows us to answer adjacency queries. Both can be done in constant time without using lookup tables. To support the neighborhood query, we use choice dictionaries on top of the group matrices. In detail, whenever updating an edge, update the summary vector, add or follow the pointer to the non-compact representation, update it including the choice dictionary and the degree counter.

After $O(n)$ operations the lookup table is computed and, in the same time, the entries in the non-compact matrices are moved to the compact submatrices.

The array A requires $n^2(\log n)/((\log^{1-\delta} n)(\log n)) = O(n^2/\log^{1-\delta} n)$ bits, which is negligible. Moreover, a group matrix with the choice dictionaries requires $O(\log^2 n)$ bits to be stored. Thus, after $O(n)$ operations, the total extra space usage is $O(n \log^2 n)$ bits, which easily fits in the extra buffers of size $\Omega(\log \binom{n^2}{m} - n \log n) = \Omega(n^2/\log^{1-\delta} n - n \log n)$ bits after $O(n)$ operations.

▶ **Theorem 3.** *A graph with n vertices can be stored with* $\log_2\binom{n}{m} + O(n^2/\log^{1-\delta} n)$ *bits supporting edge updates as well as adjacency and degree queries in constant time and iteration over neighbors of a vertex in constant time per neighbor where* $m = n^2/\log_2^{1-\delta} n$ *and* $\delta > 1/2$. *A startup time for constructing tables is not necessary.*

Note that, if the whole memory of a graph representation must be allocated in the beginning, i.e., if we do not allow a change of the space bound during the edge updates, then our representation of the dynamical graph (with possibly $n^2/\log_2^{1-\delta} n$ edges) is also succinct.

4 Extra Space for Dynamic Arrays

We now extend our framework to dynamic arrays. The data structure $\mathcal{D}$ (Definition 1) of Katoh and Goto is not dynamic, but assume it is. Then the unwritten area of $\mathcal{D}$ may become larger and smaller and with it the size of the array extra array E (Theorem 2).

Since we start indexing the words belonging to E from the end of $\mathcal{D}$, changing the size of $\mathcal{D}$ will change the index of the words in E. If we start indexing E from the beginning of the unwritten area, all indices change every time the unwritten area shrinks.

To realize an indexed array we introduce a fixed-length array F. To handle the shrink operation, we store F strictly behind the written area so that there is nothing to do if we shrink or expand the size of $\mathcal{D}$. Recall that b is the size of a block words and m is the number of write operations. If the written area grows, we lose $b-2$ unused words of the unwritten area and therefore $b-2$ used words in F. As long as there are unused words in the unwritten area behind F we move the $b-2$ words that we would lose to the first unused words behind F. This will rotate F inside the unwritten area of D.

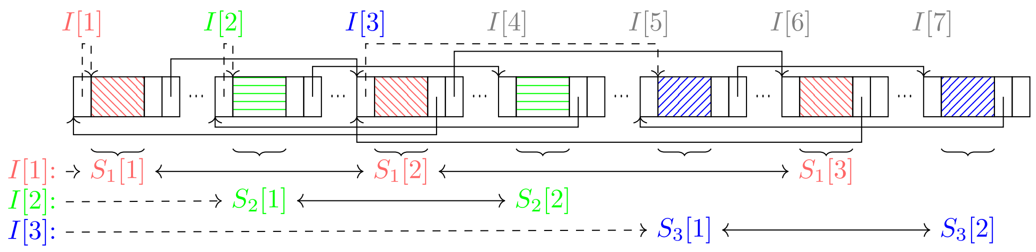
▶ **Lemma 4.** *We can store a fixed-length array F consisting of ℓ words in the unused space of $\mathcal{D}$ as long as $\ell \le (n/b - m - 1)(b-2) \le (|U|-1)(b-2)$. If the condition is violated, then F is destroyed.*

Proof. Let the size of F be smaller than the unused space of $\mathcal{D}$. Whenever the user writes an additional word in $\mathcal{D}$, m increases and the number of the unused words becomes less. Therefore, before extending the written area by one block, we move the $b-2$ words in that block to the unused space of $\mathcal{D}$ behind F. We use the last block in the unwritten area to store the size and a counter to calculate the rotation. Thus, we have one block less in contrast to Theorem 2. F can be of size $\ell \le (n/b - m - 1)(b-2) \le (|U|-1)(b-2)$. ◀

For some applications it is interesting to have a dynamic extra storage even if $\mathcal{D}$ is dynamic. We introduce a dynamic set in Lemma 5 that can be stored inside the unwritten area of $\mathcal{D}$ and supports the operations ADD, REMOVE, and ITERATE. The last operation returns a list of all elements in the data structure. The size of dynamic extra storage is dynamically upper bounded by $|U|$.

▶ **Lemma 5.** *We can store a dynamic (multi-)set of maximal $\ell \le (|U|-1)(b-2)$ elements in the unused space of $\mathcal{D}$. If the condition is violated by write operations of $\mathcal{D}$, elements of the set are removed until the condition holds again.*

Proof. We store a dynamic list strictly behind the written area and shift it cyclically as in the proof of Lemma 4. The difference here is that we have no fixed order so that we can store new elements simply at the beginning of the unused words of $\mathcal{D}$ and increase the size ℓ. The removal of an element may create a gap that can be filled by moving an element and decreasing ℓ. ◀



$I[1]$
$I[2]$
$I[3]$
$I[4]$
$I[5]$
$I[6]$
$I[7]$
$I[1]$: $S_1[1]$ $S_1[2]$ $S_1[3]$
$I[2]$: $S_2[1]$ $S_2[2]$
$I[3]$: $S_3[1]$ $S_3[2]$

Figure 3 The figure shows a dynamic family consisting of three dynamic sets. The first element of each set can be found in section to which $I[1]$, $I[2]$ and $I[3]$, respectively, points.

For our implementation described in Section 5 we require a (multi-)set with *direct access* to the elements. We can use the position in D as an address for direct access, but this requires that the user implements a *notification function*, which is called by our data structure whenever an element in the list changes its position to inform the user.

▶ **Fact 6.** *Having two chained blocks B and $d(B)$ and a suitable block size b, we can rearrange the user data in the two blocks such that we can store $O(1)$ extra words (information as, e.g., pointers) in both B and in $d(B)$.*

We next show that an algorithm can use several extra data structures in parallel.

▶ **Lemma 7.** *For a $b \geq 2$, $\mathcal{D}$ can have $k \in \mathbb{N}$ extra data structures in parallel where each is of size at most $|U|\lfloor (b-2)/k \rfloor$.*

Proof. Every block of the unwritten area has $b-2$ unused words and every unused word can be used for another data structure. ◀

As we see later, it is useful to have several dynamic sets of elements that are all the same size. The sets can also be (multi-)sets.

▶ **Corollary 8.** *For $b > 2$, $\mathcal{D}$ can store a dynamic family $\mathcal{F}$ consisting of dynamic sets where the sets have in total at most $\lfloor (|U|-3)(b-2)/(s+3) - 1 \rfloor$ elements each of size s.*

Proof. We use the unused words in the unwritten area of $\mathcal{D}$ and partition it into sections of $s+3$ words. Every first word of a section is used for an array I, then s words are used to store an element of a set, and the last two words store pointers of a doubly linked list that connect the element in a doubly-linked list with all other elements in the set. $I[i]$ points to the an element of the ith set. The element is stored in one section. Using the pointers at the end of each section, we can find the next element of the set. The realization is also sketched in Figure 3. The unused words in the last 3 blocks (possibly, fewer blocks suffice) of the unwritten area are used to store some constants: the size t of the written area, the initial value of $\mathcal{D}$, some parameter $p \in \mathbb{N}$, the number of sets in $\mathcal{S}$, and the total number q of items over all sets. Using q we can simply find the section of a new element.

If the written area of $\mathcal{D}$ increases, we have to start moving the first section to a new unused section. We store the position of the new section in p. By knowing t we know how much of the first section has been already moved and where to find the information of the first section. ◀

5 Dynamical Initializable Array

In the next three subsections, we provide implementations to make the in-place initializable array dynamic. Therefore, we extend the initializable array $\mathcal{D}$ by the following two functions to change the current size of the array.

- INCREASE$(n_{\text{old}}, n_{\text{new}}, \text{initv})$ $(n_{\text{old}}, n_{\text{new}} \in \mathbb{N})$: Sets the size of $\mathcal{D}$ from n_{old} to n_{new}. All the elements behind the n_{old}th element in $\mathcal{D}$ are initialized with initv, the initial value of $\mathcal{D}$ defined by the last call of INIT.
- SHRINK$(n_{\text{old}}, n_{\text{new}})$ $(n_{\text{old}}, n_{\text{new}} \in \mathbb{N})$: Sets the size of $\mathcal{D}$ from n_{old} to n_{new}.

In our implementation we handle the array $\mathcal{D}$ differently, according to its size. Recall that w is the word size. As long as $\mathcal{D}$ consists of $n' = O(w)$ elements, we use a bit vector stored in $O(1)$ words to know which words are initialized. On the word RAM we can manipulate it in constant time. To have immediate access to the bit vector we store it in the beginning of the array and move the data located there to the first unused words. If we increase the size of $\mathcal{D}$ to more than $\omega(w)$ elements, we keep the bit vector in some unused words until the first n' words of $\mathcal{D}$ are completely initialized. Having such a bit vector we assume that this is taken into account whenever there is a reading or writing operation, but do not mention it explicitly in the rest of the paper.

If the array $\mathcal{D}$ consists of at least $\omega(w)$ elements, we can not use the solution with the bit vector. Instead, we use chains so that we can distinguish between initialized and uninitialized blocks, even after several shrink and increase operations. We assume in the following that $\mathcal{D}$ consists of $\omega(w)$ elements.

5.1 Increasing the size of $\mathcal{D}$

An increase of the size of $\mathcal{D}$ means allocating additional memory that gives us several new blocks inside the unwritten area. These blocks may contain arbitrary values and these values can point at each other such that they create a so-called unintended chain between a block B_w (written area) and a block B_u (unwritten area). Katoh and Goto *break such unintended chains* by creating a self-chain (pointer at its own position) with B_u whenever they write a value to a block of the written area. Because the increase of $\mathcal{D}$ may be arbitrary, we can not destroy such chains in constant time.

To support large increases of $\mathcal{D}$, we eliminate the possibility of unintended chains by introducing another kind of chain, called *verification chain*. To distinguish the two kinds of chains we name the chain introduced in Section 2 a *data chain*. We define a chain between two blocks B_w and B_u as *intended* exactly if B_w has also a verification chain, otherwise as *unintended*.

To use the verification chain, we first distribute the user data among two chained blocks such that each block of the written area has an unused word (Fact 6). Let L be a dynamic list with direct access stored in $\mathcal{D}$ (Lemma 5). Whenever the block B in the written area has a data chain with a block $d(B)$, we additionally require that it also must have a verification chain with an element of the list L (Figure 4). We denote by $v(B)$ a pointer to the element in L chained with B and use an unused word of B to store $v(B)$. Recall that, whenever the unwritten area shrinks, some elements in L change their position in D. To ensure the validity of the verification chains, we use the notification function of L to update the pointer $k \in \mathbb{N}$ in the effected blocks. The verification pointers in L, stored behind the written area, are not affected by increasing the size of $\mathcal{D}$.

Finally, note that L has enough space to store all verification pointers since only one verification chain is required for each block in the unwritten area, i.e, L is of size $O(|U|)$.

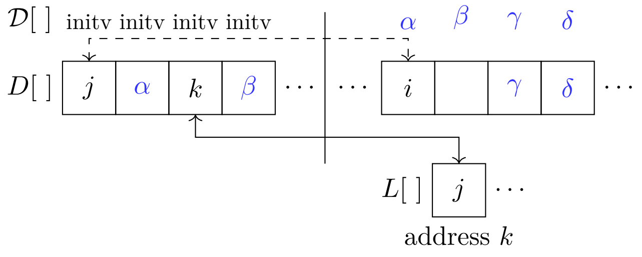

$\mathcal{D}[\,]$
initv initv initv initv
α β γ δ
$D[\,]$
j α k β $\cdots$
$\cdots$ i γ δ $\cdots$
$L[\,]$ j $\cdots$
address k

Figure 4 The block B_w (left) has a verification chain with an element of a dynamic list L to verify that the data chain between block B_w and B_u (right) is intended.

5.2 Shrinking the Size of $\mathcal{D}$ at most $O(w)$ Times

Shrinking the size of $\mathcal{D}$ means to free some memory that may be used to store information. We distinguish between two cases. In the first case we shrink the size of $\mathcal{D}$ so much that L is destroyed. In this case we make a *full initialization* of $\mathcal{D}$ as follows.

Since we lose the ability to check for unintended data chains, we first destroy them by iterating over the verification list L, following its pointer to a block B_w, checking if the block $B_u = d(B_w)$ is behind the new size of $\mathcal{D}$. If it is, we initialize B_w with the initial value. Writing the initial value into B_w may created an unintended data chain, which we break. Now the verification list is superfluous and we can iterate over the unwritten area and check for a data chain. If there is one, we move the user values from $d(B)$ into B and initialize $d(B)$, otherwise we initialize B. After the iteration, we set $D[0] = 1$.

Since L will be destroyed by the shrinking operation, the size $|U|$ of the unwritten area behind the shrinking is bounded by $|L|$. Thus, by using the potential function $\psi = (c \cdot \text{length}$ of $L)$ for some constant $c \geq 2$, the amortized cost of the WRITE operation increases by at most $c = O(1)$ and we can easily pay for the full initialization.

In the second case, we reduce the size of $\mathcal{D}$ such that the verification list L is still completely present in $\mathcal{D}$. In this case, all blocks of the written area remain and they still have a verification chain. However, the data chain can point outside of $\mathcal{D}$. If we subsequently re-increase $\mathcal{D}$, some blocks may still have a data chain and also a verification chain. This violates our definition of the INCREASE operation.

To resolve this problem we invalidate the data chains of all blocks outside $\mathcal{D}$ as follows. As long as there is no shrinking operation, we store the same *version number* $v \in \mathbb{N}$ to all data chains. More exactly, let B be a block in the unwritten area chained with a block $d(B)$ in the written area. Then we store the version inside an unused word of $d(B)$ using Fact 6.

Before we execute a shrink operation, we set n_v to the current size of $\mathcal{D}$ and remember it as the size for the current version v. With the shrink operation, we increment the version number by one. Let v' be the version of the chain between B and $d(B)$. We call the chain *valid* if B is before the boundary $n^*_{v'} = \min\{n_v | v \geq v'\}$. Note that $n^*_{v'}$ is the minimal boundary that $\mathcal{D}$ ever had after introducing version number v'.

We obtain the boundaries n^*_v from a data structure M that provides an operation to add the new size of $\mathcal{D}$ after a shrink operation and another operation MINBOUND to check if a chain is valid, i.e., if one endpoint of the chain is or was outside of $\mathcal{D}$. For later usage, M also supports an operation REMOVE.

- ADD(n) $(n \in \mathbb{N})$: Increments an internal version counter v by one and set the boundary $n^*_v = n$. All boundaries n^*_i $(i \in \{1 \dots v-1\})$ larger than n are overwritten by n.
- REMOVE(j) $(0 \leq j \leq v)$: Decrements v by j and removes the boundaries of the largest j versions.
- MINBOUND(v) $(0 \leq v < w)$: Returns n^*_v, the minimal boundary for v.

▶ **Lemma 9.** *M can be implemented such that it uses $O(w)$ words,* ADD *runs in amortized constant time, and* MINBOUND *and* REMOVE *run in constant time as long as there are only $O(w)$ versions.*

Proof. We have a version counter v^* and a table T where we store the initial boundary for each version. Moreover, we use a stack S that additionally allows us to access the elements of the stack directly and the data structure R from Pătraşcu and Thorup [25]. All three data structures can be implemented using a fixed size array and stored in extra buffers (Lemmas 4 and 7). R consists of all versions v with $n_v = n^*_v$. The stack S store the boundaries of the versions in R in ascending order from the bottom to the top. For a technical reason, S stores also the version with each boundary.

To answer the MINBOUND operation for a version v, we first determine the successor v' of v in R and then return $n^*_v = T[v']$.

Assume that a new boundary $n_v = n$ is added to M for a next version v. Set $T[v] = n$. Before adding a new boundary n_v to S we remove all boundaries from S as long as the top of S is larger than n_v. Whenever removing such a boundary, we remove the corresponding version from R. Finally, we add the new boundary to S and to R. By doing this, we remove all boundaries that are larger than n_v. Now all the versions of these boundaries get n_v as their new boundary.

Finally, it remains to check the running time. We use the potential function $\phi = |S|$ with $|S| \leq w$ being the current size of S. Determining the boundary for a specific version requires looking into R and to check one word in S and T. This does not change ϕ and runs in $O(1)$ time. If the running time for adding or removing versions is not constant, then in all except a last iteration, we remove an element from the stack. Thus, the decrease of ϕ pays for this. ◀

▶ **Corollary 10.** *For every data chain, we can check in constant time if it is valid or not.*

Proof. A chain between some blocks B and $d(B)$ is valid exactly if $d(B) \leq$ MINBOUND(v) is true, where v is the version of the chain and $d(B)$ is the position of the block that belongs to the unwritten area. ◀

Whenever a block in the unwritten area has an invalid chain (caused by shrinking and re-increasing $\mathcal{D}$; determined by MINBOUND), we return the initial value for all words of the block. We want to remark that the structure M can maintain only $O(w)$ versions since it uses a dictionary from Pătraşcu and Thorup [25]. The dictionary is dynamic and supports modification and access in constant time, but only for $O(w)$ entries.

We have to make sure that we have only $O(|U|)$ chains (counting both valid and invalid chains) so that we are able to store them. The problem is that a shrink can invalidate many chains. Therefore, we have to *clean-up* our data structure $\mathcal{D}$ as follows: Whenever we add a chain, we check the validness of three old chains, i.e., chains with an old version number that are stored at the beginning of the unwritten area in L. Invalid chains are removed whereas valid chains are assigned to the current version. To have the time for the clean-up, we also modify the shrink operation such that, after each shrink operation, we make sure that we store at most $|U|/2$ chains. If this is not the case, we run a full initialization. By assuming that every insert into $\mathcal{D}$ pays a coin, this can be done in amortized constant time since the number of chains that we have is bounded by the number of insert operations.

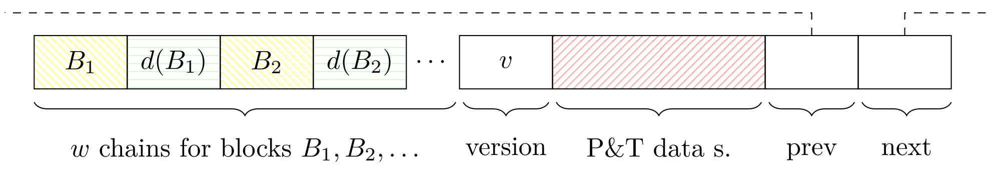


Figure 5 A subgroup that is embedded as an element of the dynamic doubly linked list containing several block positions of a version v that are indexed with a dictionary.

5.3 The General Case

The goal in this section is to limit the number of versions to $O(w)$ such that the data structure M from Lemma 9 can be always used. We achieve this goal by *purifying* the chain information, i.e., for some old version v', we iterate through the chains with a version larger than v', remove invalid data chains and store the remaining chains under v'. Finally, we take v' as the current version. We so remove all versions larger than v', which now can be reused.

Since we have no data structure to find invalid chains out of a large amount of all chains, we partition the chains of each version into small subgroups and use also here the dictionary from Pătraşcu and Thorup [25] to index the block positions B and $d(B)$ that represent the chain. The dictionary from the subgroup with a version v allows us to find a block that lies behind a boundary n^*_v in constant time as long as such a block exists.

In detail, a subgroup always has space for $\Theta(w)$ chains (block positions), the version v, and a data structure from Pătraşcu and Thorup [25] (Figure 5). To organize the subgroups we use a dynamic family S from Corollary 8. For each version v, we create a dynamic set in S. Each set consists of all subgroups of the same version. Instead of having a verification chain with a list L, each chained block in the written area has now a pointer to its subgroup. If the block is indexed in the subgroup, then its chain is verified.

As in the previous section we use the data structure M to store and check the boundaries. Additionally, we store the version v and a table T. The table is used to store, for each version, the number of chains that are currently used. The updates of T are not described below explicitly. Based on the information in T we determine how many versions we can purify. The details of the PURIFY operation are described on the next page.

Note that M and T are of $O(w)$ words and the size of S is linear in the size of the number of chains and thus bounded by $O(|U|)$. Choosing the block size b large enough, but still $b = O(1)$, we can guarantee that all data structures fit into $|U|$ unless $|U| = O(w)$. By running the clean-up of the last section, M and T can be removed if to many writing operations shrink the unwritten area of $\mathcal{D}$ so much that $|U| = O(w)$.

- INITIALIZE(n, initv) Allocate $nw + 1$ bits. Partition the array into blocks of size $b = 6$. If n is not a multiple of b, initialize less than b words and treat them separately. Use the last block(s) to store the threshold $t = 0$, initv, and $v = 1$ in the unused words. Initialize the data structure M and S, store their internally required single words also in the last block(s) and their sets and lists in parallel (Lemma 7).
- READ(i): If $D[0] = 1$, return $D[i]$. Else, check to which area $B = \lfloor i/b \rfloor$ belongs. If B is inside the written area, check if B is verified. If it is, return initv, otherwise $D[i]$.
  If B is inside the unwritten area, check if B is chained with a block $d(B)$. If it is not, return initv, otherwise proceed with checking if $d(B)$ is verified. If it is not, return initv,

otherwise follow the verification pointer and read the version v out of the subgroup. Call M.MINBOUND(V) and return initv if it returned a boundary that is larger than the block position B, otherwise return the right word out of B and $d(B)$.

- WRITE(i, x): If $D[0] = 1$, write at $D[i]$. Otherwise, clean-up three chains as described in the last subsection. Then check to which area $B = \lfloor i/b \rfloor$ belongs. If B is inside the written area, check if B is verified. If it is not, write at $D[i]$ directly. Otherwise, the block may have a data chain with a block $d(B)$. If so, unchain it (like in [17]) by expanding the written area that gives us a new unused block that we use to relocate all values and chains from B. Correct also the chains in the subgroups. If not, just remove B and $d(B)$ from its subgroup. In both cases, B becomes an unused block afterwards. We initialize it and write at $D[i]$ directly. If B is inside the unwritten area, check if B is chained with a block $d(B)$. If it is, check if $d(B)$ is verified and if its chain belonging to a version v is valid (test M.MINBOUND$(v) \geq B$). If all is true, then write x at the right position of B and $d(B)$. If the chain is invalid, delete it from its subgroup and move it into the latest subgroup of the current version. Finally, initialize B and write x as described above.

  But if B is not chained, expand the written area and chain it with the new block. Write both blocks inside the latest subgroup of the current version. Set a verification pointer to the subgroup where the chain is stored. As before, initialize B and write x as described above.

  In all cases, whenever the unwritten area disappears, set $D[0] = 1$.
- INCREASE(n_{old}, n_{new}, initv): If $D[0] = 0$, then copy the words of the last block(s) into the new last block(s). Otherwise, initialize the required data structures as described in INITIALIZE, but set the initial values for the threshold t to $t = \lfloor n_{\text{old}}/b \rfloor$. If $|U| = O(w)$, use the bit vector solution described in the beginning of Section 5.
- PURIFY(n_{new}): Iterate from the current version $v = \lceil w \rceil$ in S down and add up the number of chains stored under a version. Stop at the first version v' that has fewer chains as twice the total number of chains of all versions visited before. Consider versions $v' + 1$ to v and iterate over all subgroups with that version. In each subgroup, check for an invalid chain (B, $d(B)$) by using M, remove it from the subgroup, from the index of the subgroup, and initialize B with initv and repeat this on the subgroup until it has no invalid chains. In the special case where the subgroup has less than w chains, clear the subgroup by moving all chains into the latest subgroup of the version v'.

  Then, change the version number of this subgroup to v', unlink it from its old set and link it into the set of v'. When finished, set the current version number v to v'.
- SHRINK($n_{\text{old}}, n_{\text{new}}$): If the new size of $\mathcal{D}$ either cuts the space used to store the data structures S or M or leaves less than $\Theta(w)$ unused words in the unwritten area, run the full initialization.

  If we do not fully initialize $\mathcal{D}$, we proceed as follows: If we have $\lceil w \rceil$ versions, call PURIFY. Otherwise, create a new set inside S and increment the current version number v by one and check if the last used subgroup has less than w entries. If so, we reuse this subgroup by removing all invalid chains. (In PURIFY we already described the removal.)

Who pays for the purification? The idea is to give a gold coin to each previous set in S, whenever we insert an element. Whenever a set of size x has at least $x/2$ coins, we clean up the set and all sets with a larger version number. Algorithmically we can not check fast enough if a set already has enough coins. Therefore, we wait with the purification until we have $\lceil w \rceil$ versions and check then the condition for every version from the largest version to the smallest until we found the first version with enough coins.

▶ **Theorem 11.** *There is a dynamic array that works in-place and supports the operations* INITIALIZE*,* READ*,* WRITE *as well as* INCREASE *in constant time and* SHRINK *in amortized constant time.*

Proof. It remains to show the running times of the operations. Take ψ as the total number of verification chains similar to Subsection 5.2, c_i as the number of chains with the version $i \in \{0, \ldots, v-1\}$ and $g = \sum_{i=0}^{v-1} \frac{6i \cdot c_i}{w}$. Moreover, choose τ as the total number of missing elements such that all subgroups are full and $f = (1 - D[0]) \max\{0, 2w - |U|\}$. We use the following potential function $\phi = \psi + v + g + \tau + f$. Note that g corresponds to the usage of the gold coins and f is non-zero only if there are very few blocks in the unwritten area U, but $\mathcal{D}$ is not completely initialized.

- INITIALIZE: We have to set the threshold t and the current version in $O(1)$ time. Thus, $\phi = v = 1$ – note that $f = 0$ by using the bit-vector solution if necessary.
- READ: Checking chains, reading the version number and calling the MINBOUND function of M can be done in constant time and ϕ does not change.
- WRITE: We possibly have to check if a block is chained, verified and valid. We also may create a chain, a verification and insert it into S. All these operations require $O(1)$ time. In total, the number of chains may increase by one $\Delta\psi = O(1)$, $\Delta g = \frac{6v}{w}$ with v being the current version. Thus, $\Delta\phi = O(1)$.
- INCREASE: Copying a few blocks runs in $O(1))$ time and $\Delta\phi = 0$ since $\Delta f = 0$.
- PURIFY: Whenever we run PURIFY we remove the last $v - i$ versions for the largest i with $v_i/2 \leq y := \sum_{i+1}^{v-1} c_i$ is valid. Thus, we first have to search for version i, i.e, we consider $v - i$ versions. Moreover, we then have to iterate over $3y$ chains in $6y/w$ subgroups – a subgroup is always at least half full. In each subgroup we spend $1 + d$ time, where d is the number of deleted chains in each subgroup. This can be done in at most $(v - i) + 6y/w + d^*$ time units where d^* is the total number of deleted chains. In addition, we may have deleted half empty subgroups and moved their chains to the latest subgroup. This can be done in $O(m)$ time if m is the number of moved chains.

  We can pay for the running time since the value of the potential function decreases as follows. By deleting versions larger than i, $\Delta v = -(v - i)$. Since we change the version of all subgroups with a version larger than i to i, i.e., we shrink the version of y chains by at least one, $\Delta g = -6y/w$. In addition, $\Delta(\psi + \tau) \leq -2d^* + (d^* - m) = -d^* - m$ where the $-m$ comes from the fact that half empty subgroups disappear, i.e, are not taken into account in τ. To sum up, the amortized running time is $O(1)$.
- SHRINK: In Subsection 5.2, we already analyzed the case that $\mathcal{D}$ is fully initialized – the only change is that we have further parts in the potential function, but their change is either zero or negative. Otherwise, we have to delete invalid chains in the latest subgroup and update it to the new version, which can be done in $O(1)$ amortized time since $\Delta\psi$ drops linear in the number of deleted chains. Moreover, we have to add a new version to M and possibly delete several older versions. All this can be done in $O(1)$ amortized time. And finally, we may run a PURIFY; again in $O(1)$ amortized time.

We thus have shown all running times as promised in the theorem. ◀

References

1 Tetsuo Asano, Kevin Buchin, Maike Buchin, Matias Korman, Wolfgang Mulzer, Günter Rote, and André Schulz. Reprint of: Memory-constrained algorithms for simple polygons. *Comput. Geom. Theory Appl.*, 47(3, Part B):469–479, 2014. doi:10.1016/j.comgeo.2013.11.004.

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 Luis Barba, Matias Korman, Stefan Langerman, Rodrigo I. Silveira, and Kunihiko Sadakane. Space-time trade-offs for stack-based algorithms. In *Proc. 30th International Symposium on Theoretical Aspects of Computer Science (STACS 2013)*, volume 20 of *LIPIcs*, pages 281–292. Schloss Dagstuhl – Leibniz-Zentrum für Informatik, 2013. doi:10.4230/LIPIcs.STACS.2013.281.

4 Paul Beame. A general sequential time-space tradeoff for finding unique elements. *SIAM J. Comput.*, 20(2):270–277, 1991. doi:10.1137/0220017.

5 Jon Bentley. *Programming Pearls.* ACM, New York, NY, USA, 1986.

6 Sankardeep Chakraborty, Anish Mukherjee, Venkatesh Raman, and Srinivasa Rao Satti. Frameworks for designing in-place graph algorithms. *CoRR*, abs/1711.09859, 2017. arXiv:1711.09859.

7 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.

8 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.

9 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.

10 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.

11 Arash Farzan and J. Ian Munro. Succinct encoding of arbitrary graphs. *Theor. Comput. Sci.*, 513:38–52, 2013. doi:10.1016/j.tcs.2013.09.031.

12 Torben Hagerup and Frank Kammer. Succinct choice dictionaries. *Computing Research Repository (CoRR)*, arXiv:1604.06058 [cs.DS], 2016. arXiv:1604.06058.

13 Torben Hagerup and Frank Kammer. On-the-fly array initialization in less space. In *Proc. 28th International Symposium on Algorithms and Computation (ISAAC 2017)*, volume 92 of *LIPIcs*, pages 44:1–44:12. Schloss Dagstuhl – Leibniz-Zentrum für Informatik, 2017. doi:10.4230/LIPIcs.ISAAC.2017.44.

14 Torben Hagerup, Frank Kammer, and Moritz Laudahn. Space-efficient Euler partition and bipartite edge coloring. *Theor. Comput. Sci., to appear*, 2018. doi:10.1016/j.tcs.2018.01.008.

15 Frank Kammer, Dieter Kratsch, and Moritz Laudahn. Space-Efficient Biconnected Components and Recognition of Outerplanar Graphs. In *Proc. 41st International Symposium on Mathematical Foundations of Computer Science (MFCS 2016)*, volume 58 of *LIPIcs*, pages 56:1–56:14. Schloss Dagstuhl – Leibniz-Zentrum für Informatik, 2016. doi:10.4230/LIPIcs.MFCS.2016.56.

16 Frank Kammer and Andrej Sajenko. Linear-time in-place DFS and BFS in the restore model. *Computing Research Repository (CoRR)*, arXiv:1803.04282 [cs.DS], 2018. arXiv:1803.04282.

17 Takashi Katoh and Keisuke Goto. In-place initializable arrays. *Computing Research Repository (CoRR)*, arXiv:1709.08900 [cs.DS], 2017. arXiv:1709.08900.

18 Veli Mäkinen and Gonzalo Navarro. Dynamic entropy-compressed sequences and full-text indexes. *ACM Trans. Algorithms*, 4(3), 2008. doi:10.1145/1367064.1367072.

19 Kurt Mehlhorn. Data structures and algorithms 1: Sorting and searching. In *EATCS Monographs Theor. Comput. Sci.*, 1984.

20 J. Ian Munro, Rajeev Raman, Venkatesh Raman, and S. Srinivasa Rao. Succinct representations of permutations and functions. *Theor. Comput. Sci.*, 438:74–88, 2012. doi:10.1016/j.tcs.2012.03.005.

21 Gonzalo Navarro. Spaces, trees, and colors: The algorithmic landscape of document retrieval on sequences. *ACM Comput. Surv.*, 46(4):52:1–52:47, 2014. doi:10.1145/2535933.

22 Rasmus Pagh. Low redundancy in static dictionaries with constant query time. *SIAM J. Comput.*, 31(2):353–363, 2001. doi:10.1137/S0097539700369909.

23 Jakob Pagter and Theis Rauhe. Optimal time-space trade-offs for sorting. In *Proc. 39th Annual IEEE Symposium on Foundations of Computer Science (FOCS 1998)*, pages 264–268. IEEE Computer Society, 1998. doi:10.1109/SFCS.1998.743455.

24 Mihai Pătraşcu. Succincter. In *Proc. 49th Annual IEEE Symposium on Foundations of Computer Science (FOCS 2008)*, pages 305–313. IEEE Computer Society, 2008. doi:10.1109/FOCS.2008.83.

25 Mihai Pătraşcu and Mikkel Thorup. Dynamic integer sets with optimal rank, select, and predecessor search. In *Proc. 55th IEEE Annual Symposium on Foundations of Computer Science, (FOCS 2014)*, pages 166–175. IEEE Computer Society, 2014. doi:10.1109/FOCS.2014.26.