Space-efficient Basic Graph Algorithms

Amr Elmasry[1], Torben Hagerup[2], and Frank Kammer[2]

1 **Department of Computer Engineering and Systems**
Alexandria University, Alexandria 21544, Egypt
elmasry@mpi-inf.mpg.de

2 **Institut für Informatik, Universität Augsburg**
86135 Augsburg, Germany
{hagerup,kammer}@informatik.uni-augsburg.de

Abstract

We reconsider basic algorithmic graph problems in a setting where an n-vertex input graph is read-only and the computation must take place in a working memory of $O(n)$ bits or little more than that. For computing connected components and performing breadth-first search, we match the running times of standard algorithms that have no memory restrictions, for depth-first search and related problems we come within a factor of $\Theta(\log\log n)$, and for computing minimum spanning forests and single-source shortest-paths trees we come close for sparse input graphs.

1998 ACM Subject Classification F.2.2 Nonnumerical Algorithms and Problems, G.2.2 Graph Theory

Keywords and phrases graph algorithms, depth-first search, single-source shortest paths, register input model

Digital Object Identifier 10.4230/LIPIcs.STACS.2015.288

1 Introduction

Motivated on the one hand by the increased prevalence of huge data collections ("big data"), and on the other hand by the emergence of small mobile devices and embedded systems that cannot be equipped with very large memories, recent years have seen a surge of interest in data structures and algorithms that treat memory space as a scarce resource.

The classical area of Turing machines that operate in logarithmic space is still very active [17, 18, 21, 22]. Practical concerns, however, lead us to focus on algorithms that run in near-linear time. Even for the fundamental s-t connectivity problem on directed graphs, the most space-efficient algorithm, due to Barnes et al. [7], needs $n/2^{O(\sqrt{\log n})}$ bits of memory when required to run in polynomial time. The bound is only slightly sublinear, and a nearly matching lower bound is known for the so-called NNJAG model [20]. In addition, Tompa [33] showed that certain natural algorithmic approaches to the problem require superpolynomial time if the number of bits available is $o(n)$. It therefore seems reasonable to accord algorithms that operate on general n-vertex graphs approximately n bits of working memory. In return, we would hope to match the time bounds of standard graph algorithms or to come close. For this to be possible, it is necessary to replace the Turing machine by a model closer to computing practice, and a number of such models have been proposed. Common to all of them is that access to the input is restricted in some way. In the *multi-pass streaming* model [27], the input can only be accessed in a purely sequential fashion, and the main goal is to minimize the number of passes over the input. Another model [9] allows the input items to be permuted but not destroyed, and in the

32nd Symposium on Theoretical Aspects of Computer Science (STACS 2015).
Editors: Ernst W. Mayr and Nicolas Ollinger; pp. 288–301

Leibniz International Proceedings in Informatics
Schloss Dagstuhl – Leibniz-Zentrum für Informatik, Dagstuhl Publishing, Germany


restore model [13], the input may be temporarily modified during a computation, but must be restored to its original state.

In this paper we employ the *register input model* of Frederickson [24]. It features a read-only input memory and a write-only output medium. The computation proper takes place in a working memory of limited size. When stating that a problem can be solved with a certain number of bits, what we mean is that the working memory comprises that many bits. The input and working memories are divided into words of w bits for a fixed parameter w, arithmetic and logical operations on w-bit words take constant time, and random access to the input and working memories is provided. In the context of inputs of n words, we assume, as is common, that $w = \Theta(\log n)$ and, in particular, that w is large enough to allow all words in the input and working memories to be addressed.

A number of results are known for the register input model. For the fundamental problem of sorting n items, Pagter and Rauhe [29] described a comparison-based algorithm that, for every given s with $\log n \leq s \leq n/\log n$, runs in $O(n^2/s)$ time using $O(s)$ bits, and a matching lower bound of $\Omega(n^2)$ for the time-space product was established by Beame [8] for the strong *branching-program model*. Other researchers have considered selection [10, 12, 23, 24, 28, 31] and various problems in computational geometry [2, 4, 5, 6, 11, 16]. With one exception, discussed below, we are not aware of previous work that reduces the working space needed to process n-vertex graphs below $\Theta(n \log n)$ bits with only a modest penalty in the running time.

1.1 New Results

We describe a number of algorithms for the register input model, all of which input a directed or undirected graph (plus, possibly, other items). When discussing graph algorithms below, we always use n and m to denote the number of vertices and the number of edges, respectively, in the input graph.

A focal point of our work is depth-first search (DFS) (Section 3) and its applications (Section 4). We first show that a DFS can be carried out in $O((n + m) \log n)$ time with $(\log_2 3 + \epsilon)n$ bits, for arbitrary fixed $\epsilon > 0$. A very similar result was found independently by Asano et al. [3]. They need cn bits, for an unspecified constant $c > 2$, or $\Theta(mn)$ time, however. Relaxing the space bound to $O(n)$ bits, we can perform the DFS in just $O((n + m) \log \log n)$ time. We also show how to achieve linear time with $O(n \log \log n)$ bits and how to interpolate between the two latter results with the same time-space product. Our main technique can be viewed as employing an "approximate runtime stack": With just $O(\log \log n)$ bits rather than $\Theta(\log n)$ bits for each vertex on the stack, we store only an approximation of its stack entry and only an approximation of its position on the stack and show how to execute the DFS in the face of the resulting uncertainty.

Some applications of DFS process the output of a DFS in reverse order. This is rarely highlighted, since reversing a sequence is implicit when the whole sequence fits in memory. When this is not the case, however, the operation can become a bottleneck. While reversing a sequence in general may be more expensive, we show how to run a DFS in reverse with only a modest penalty of $O(n \log \log n)$ additional bits. This allows us to compute topological sortings and strongly connected components in linear time with $O(n \log \log n)$ bits. Here the main technique employed is to keep enough information about a DFS to restart it in the middle and to use this repeatedly to reverse small pieces of its output, produced in reverse order, one at a time. Although the connected components of an undirected graph are usually computed by means of DFS, in Section 5 we observe that this bottleneck can be avoided and show how to compute the connected components in $O(n + m)$ time with $O(n)$ bits and how to carry out a breadth-first search within the same bounds.

Section 6 describes space-efficient versions of the algorithms of Prim and Dijkstra for computing a minimum spanning forest (MSF) and a single-source shortest-paths (SSSP) tree, respectively. To describe the algorithms, we introduce the notion of a priority queue with a *deletion budget*. A priority queue with a deletion budget uses less space than a usual priority queue, but can be used only for a certain time before it must be *refilled*, i.e., initialized anew. We give two algorithms for the MSF problem. The first one runs with $O(n)$ bits in $O(n+m \log n)$ time. The second algorithm uses more space, namely $O(n \log(2+m/n))$ bits, but matches the running time of $O(m + n \log n)$ of usual implementations of Prim's algorithm. Despite the pronounced similarities between Prim's and Dijkstra's algorithms, the SSSP problem appears more difficult in a space-efficient setting because the vertices of the SSSP tree cannot be output and forgotten as they are computed; rather, their distances from the source are needed later in the computation. We cannot store the distances, and in order to recompute them with any degree of efficiency, we must remember the SSSP tree, which needs $\Theta(n \log(2 + m/n))$ bits. While this number of bits is $O(n)$ for sparse graphs with $m = O(n)$, it degrades to $\Theta(n \log n)$ for dense graphs with $m = \Theta(n^2)$. Assume, by way of example, that the input graph is sparse and that we want to use only $O(n)$ bits. Then we can recompute the distances from the SSSP tree in batches of size $\Theta(n/\log n)$ in $O(n)$ time. Since we need to do this for $\Theta(\log n)$ batches for each of $\Theta(\log n)$ refillings of the priority queue, the total time becomes $O(m+n(\log n)^2)$. More generally, if $O(n(\log(2+m/n)+s(n)))$ bits are available, we achieve a time bound of $O(m + n \log n + n((\log n)/s(n))^2)$.

2 Preliminaries

It is customary to distinguish between adjacency matrices and adjacency lists, but not to specify the input format of a graph algorithm in any greater detail. This is because linear time and a linear number of words of working memory are sufficient to convert between any two reasonable adjacency-list representations—e.g., the edges may be reordered by means of radix sorting. In our setting, where we want to get by with $o(n \log n)$ bits of working memory, we have to be more specific about the input format.

Let $G = (V, E)$ be an input graph with n vertices and m edges. As is common, we always assume that $V = \{1, \ldots, n\}$ and that, given $u \in V$, we can access the set $N(u)$ of neighbors of u (if G is undirected) or of outneighbors of u (if G is directed). For some algorithms it suffices to be able to iterate over $N(u)$ in constant time per vertex, the archetypical functionality provided by adjacency lists. For most of our algorithms, however, we need random access to $N(u)$. More precisely, given u and an integer k with $1 \leq k \leq |N(u)|$, we need constant-time access to the kth element of $N(u)$. In such cases we will indicate that the input graph must be represented via *adjacency arrays*.

Some algorithms have additional special requirements. When we state that an adjacency-array representation of an undirected graph has *cross pointers*, what we mean is that, given a vertex u and the position in $N(u)$ of a neighbor v of u, in constant time we can find the position of u in $N(v)$. Our algorithm for computing the strongly connected components of a directed graph assumes that, given a vertex u, we have access not only to its outneighbors, but also to its inneighbors. We will formulate this by stating that the input graph must be represented with *in/out adjacency lists* or *arrays*. Our algorithm for the single-source shortest-paths problem uses in/out adjacency arrays, say, $N_{\text{in}}(u)$ and $N_{\text{out}}(u)$ for $u \in V$, and requires the arrays $N_{\text{out}}(u)$, for $u \in V$, to be sorted consistently with a linear order on V that is either the natural order $1, \ldots, n$ or is specified in the input. We will say that the input graph must be represented with *sorted adjacency arrays*. In addition, for each

$(u, v) \in E$, given u and the position of v in $N_{\text{out}}(u)$, we must be able to find the position of u in $N_{\text{in}}(v)$ in constant time—again, we will say that the representation must have cross pointers.

An inconspicuous but crucial role is played in most of our algorithms by a special case, characterized in the following lemma, of a data structure developed in [26].

▶ **Lemma 2.1.** *For every fixed $n \in \mathbb{N} = \{1, 2, \ldots\}$, there is a dictionary that can store a subset A of $\{1, \ldots, n\}$, each $a \in A$ with a string h_a of satellite data of $O(\log n)$ bits, in $O(n + \sum_{a \in A} |h_a|)$ bits such that membership in A can be tested in constant time for each element of $\{1, \ldots, n\}$, h_a can be inspected in constant time for each $a \in A$, elements with their satellite data can be inserted in and deleted from A in constant amortized time, an operation some_id that returns an (arbitrary) element of A is supported in constant time, and an operation all_ids that returns all elements of A is supported in $O(|A| + 1)$ time.*

We sometimes want to store for each vertex v in a graph with n vertices and m edges an index into the adjacency array of v. Jensen's inequality and Lemma 2.1 show that this can be done with $O(n \log(2 + m/n))$ bits.

3 Depth-First Search

A DFS of a directed graph $G = (V, E)$ steps through the vertices of G and *processes* each in turn if its processing has not already begun. The processing of a vertex $u \in V$ consists in stepping through its outgoing edges and, for each such edge (u, v), *exploring* (u, v) and, if the processing of v has not already started, processing v recursively. Every vertex is processed exactly once, and every edge is explored exactly once. When the processing of a vertex $u \in V$ starts, we say that u is *discovered*.

It is customary to use a stack to keep track of the vertices whose processing has begun, but not yet ended, with vertices that were discovered more recently appearing closer to the top of the stack. When a vertex is discovered, it is pushed on the stack, and when its processing terminates, it is again at the top of the stack, from which it is popped. Following Cormen et al. [14], we call a vertex *white* if it has not yet been discovered, *gray* if its processing is underway, and *black* if its processing has ended.

Whenever an edge (u, v) is explored, u is at the top of the stack. If the exploration of (u, v) causes v to be pushed on the stack above u, i.e., if v is white just prior to the exploration of (u, v), v becomes gray at that point and will remain gray and immediately above u on the stack until v is popped and turns black. Thus, whenever a vertex v appears immediately above another vertex u on the stack, (u, v) is an edge of E and the first edge out of u whose head is neither black nor stored below v on the stack.

As described so far, depth-first search does not do anything useful—it is just an "empty control structure". Applications of DFS therefore augment the basic scheme with additional computational steps. Such steps can be executed, e.g., at the beginning and/or at the end of each processing of a vertex and/or at the exploration of each edge. If they are phrased as application-dependent *user* procedures *preprocess*, *postprocess*, *preexplore* and *postexplore*, DFS can be expressed via the code fragment below, which denotes the outdegree of a vertex u by $\deg(u)$ and its kth outneighbor by $N(u)[k]$, for $k = 1, \ldots, \deg(u)$.

```
DFS:
  for u := 1 to n do color[u] := white;
  for u := 1 to n do if color[u] = white then process(u);
```

The procedure *process* is defined as follows:

```
process(u):
  color[u] := gray;
  preprocess(u);
  k := 1;
  while k ≤ deg(u) do
    v := N(u)[k];
    preexplore(u, v, color[v]);
    if color[v] = white then process(v);
    postexplore(u, v);
    k := k + 1;
  postprocess(u);
  color[u] := black;
```

We view the problem to be solved as that of executing the correct sequence of calls of *preprocess*, *postprocess*, *preexplore* and *postexplore*. Of course, we exclude the time and space requirements of these procedures from our resource bounds, and we often ignore them in what follows so as not to clutter the picture.

The execution of the procedure *DFS* uses $\Theta(n)$ bits of working memory for the array *color*. However, the implicit run-time stack needed to keep track of partially executed calls of *process* may require $\Theta(n \log n)$ bits. As a first step towards more space-efficient solutions, we eliminate the explicit recursion from the procedure *process* and reformulate it below to manage its own run-time stack. The latter, denoted by S, stores not just vertices, but pairs consisting of a vertex u and an integer that indicates the number of the next edge out of u to be explored. Pushing a pair (u, k) on S is written $S \Leftarrow (u, k)$, popping the top entry from S and storing its components in u and k is written $(u, k) \Leftarrow S$, and S is tested for being nonempty with $S \neq \emptyset$.

```
process(u):
  S ⇐ (u, 1);
  while S ≠ ∅ do
    (u, k) ⇐ S;
    color[u] := gray;
    if k ≤ deg(u) then
      S ⇐ (u, k + 1);
      if color[N(u)[k]] = white then S ⇐ (N(u)[k], 1);
    else color[u] := black;
```

Informally, the presence on S of an entry of the form (u, k) signals that at some point, namely when (u, k) again becomes the top entry of S, the algorithm will proceed to either process the kth edge out of u or discover that $k > \deg(u)$. In the first case, (u, k) is replaced by $(u, k + 1)$ as the top entry on S. Although this is formulated above in terms of standard stack operations as a pop followed by a conditional push, our discussion will instead pretend that it happens as a test of the value of a field in the top entry on S followed by—depending on the outcome—an increment of that value or a pop.

3.1 A Simple DFS Algorithm

An entry on S can be represented in $\Theta(\log n)$ bits and, in general, needs that much space. Since S may grow to contain as many as n entries, the algorithm stated above requires

$\Theta(n \log n)$ bits of working space. The goal in this section is to reduce the space requirements to little more than n bits while incurring only a logarithmic penalty in the time bound.

▶ **Theorem 3.1.** *For every constant $\epsilon > 0$, a DFS of a graph with n vertices and m edges can be performed in $O((n+m)\log n)$ time with at most $(\log_2 3 + \epsilon)n$ bits.*

Proof. Without loss of generality, assume below that n is larger than a certain constant. Take $\lambda = \log_2 3$. Among other data structures detailed below, we need the array *color* of n entries drawn from $\{white, gray, black\}$. It is well-known and easy to see that *color* can be realized in at most $(\lambda + \epsilon/3)n$ bits so that individual entries can be tested and set in constant time (assume without loss of generality that $3/\epsilon$ is an integer that divides n and store each group of $3/\epsilon$ consecutive color values in $\lceil \log_2(3^{3/\epsilon}) \rceil = \lceil 3\lambda/\epsilon \rceil$ bits).

Compute q as a positive integer with $q = \Theta(n/\log n)$, chosen so that $2q$ entries on S take up at most $(\epsilon/3)n$ bits. At any given time, we partition the entries on S into $O(\log n)$ *segments* as follows: The bottommost q entries form the first segment, the next q entries form the second segment, and so on, with the last segment usually containing fewer than q entries. Let us call the last (most recently pushed) entry within each segment its *trailer*.

We remember only a part S' of the full stack S. S' always consists of the one or two last (most recent) segments of S and therefore, by the choice of q, requires no more than $(\epsilon/3)n$ bits of storage. In addition, we store all trailers present on S on a separate stack T of $O((\log n)^2)$ bits. T and various simple variables together can be stored in fewer than $(\epsilon/3)n$ bits, so the total space requirements are bounded by $(\lambda + \epsilon)n$ bits.

The algorithm works with S' exactly as the usual DFS algorithm works with S (of course, additionally manipulating trailers as appropriate), except in the following two special events: (1) When S' already contains $2q$ entries and a new entry is to be pushed on S, first the older of the two segments present on S' is dropped to make room for a new segment. (2) When S' loses its last entry due to a pop but S (and hence T) is not empty, the one or two topmost segments of S are restored and placed on S', after which the normal execution resumes.

The restoration of a segment is performed as follows: First all gray vertices are recolored white. Then the DFS is restarted from the beginning, except that black vertices remain black and that the process operates *quietly*, i.e., the user procedures *preprocess*, *postprocess*, *preexplore* and *postexplore* are not executed. The restoration is continued until the top entries on S' and T coincide, at which point one (if there is only one) or two segments of S will have been restored on S'.

To see that the restoration is correct, recall that if u and v are successive vertices on S, (u, v) is the first edge out of u whose head is neither black nor stored on S below v. Thus all vertices that are pushed on S' during the restoration, except for the last trailer, will simply skip over their first outgoing edges, those that point to gray or black vertices, and push the first white vertex encountered—the correct next vertex on the stack—while coloring it gray. In particular, no vertices are ever popped, so no restoration will be called for during the restoration. The last trailer will skip over edges to gray or black vertices and reach the edge that was the next edge to be explored in the original DFS, which is resumed at that precise point.

To bound the number of restorations, consider the potential function $\Phi = \max\{q - |S'|, 0\}$, where $|S'|$ is the number of entries currently stored on S'. $\Phi = q$ initially and $\Phi \geq 0$ at all times, no push increases Φ, a pop increases Φ by at most 1, and a restoration decreases Φ from q to 0. As each of the n vertices is popped only once, the number of restorations is bounded by $(q+n)/q = O(\log n)$. It is obvious that a restoration can be executed in $O(n+m)$ time. The computation outside of restorations also runs in $O(n+m)$ time, as it is basically a standard DFS, so the total time comes to $O((n+m)\log n)$. ◀

3.2 Depth-First Search in Linear Time

This section describes a DFS procedure that works in $O(n+m)$ time and uses $O(n \log\log n)$ bits. Several notions carry over from the previous section: segments of $q = \Theta(n/\log n)$ entries on S, the stack S' that contains only the last one or two segments on S, and the restoration of the topmost segment of S when S' becomes empty. There are two main new ideas, explained in the following.

The first idea is to carry out a restoration not by restarting the DFS from scratch, but by using the trailer of the segment just below the top segment of S as a starting point (if S contains just one segment, restart the DFS from the beginning). Thus the goal is to restore just the top segment without going through the process of reconstructing the segments below it only to throw them away immediately after.

The idea expressed in the previous paragraph meets with a difficulty. Recall that in the algorithm of Section 3.1, the restoration begins by recoloring all gray vertices white, so that they are again eligible for being pushed on the stack. Such an operation would be too expensive in the present context. Besides, what we need is something different: a recoloring that is applied only to the vertices in the top segment of S, since only these should be allowed to enter S'. We achieve a similar effect by numbering the segments consecutively from bottom to top and introducing a table D with an entry for each vertex in V. Whenever a vertex $u \in V$ is pushed on S', the number of the segment that it enters is stored permanently in $D[u]$. Since all segment numbers are $O(\log n)$, D can be stored in $O(n \log\log n)$ bits. In addition, we temporarily switch the meaning of the colors white and gray for the vertices in the top segment of S for the duration of the restoration.

The restoration process is modified as follows: At each exploration of an edge (u, v), v is pushed on S' exactly if $D[v]$ indicates that v belongs to the top segment on S and v is gray. If v is pushed, it is colored white to prevent it from being pushed again later in the restoration. When the restoration is complete, the vertices in the restored top segment are all white, and they are recolored gray (their "true" color) before the original DFS resumes.

The second new idea serves to speed up the search for the correct edge out of a vertex u on the stack during a restoration. Ideally, we would like to know the integer k such that the pair (u, k) is stored on S, but we do not have the space to remember this information for all vertices. Define a vertex to be *big* if its degree exceeds m/q. For the at most q big vertices we store the relevant pairs explicitly. More precisely, the part of S maintained on T is extended to include not only the trailers, but also all pairs (u, k), where u is big. During a restoration, the part of T above and including the topmost trailer is accessed from bottom to top, in synchrony with the restoration, so that the need to search through the outgoing edges of big vertices is eliminated—the correct value of k is found in constant time. For other vertices we store a rough, $O(\log\log n)$-bit approximation of the relevant k. Compute l as a positive integer with $l = \Theta(\log n)$. For each pair (u, k) stored on S with $\deg(u) \geq 1$, we extend the table entry $D[u]$ to contain also the integer $f_u = \lfloor (k-1)/g_u \rfloor$, where $g_u = \lceil \deg(u)/l \rceil$. Informally, f_u indicates the number of groups of g_u edges out of u that have been completely explored. The restoration is changed to skip the processing of edges in such groups.

For all $u \in V$ with $\deg(u) \geq 1$, $f_u \leq (k-1)/g_u \leq (k-1)l/\deg(u) \leq \deg(u)l/\deg(u) = O(\log n)$, so $O(n \log\log n)$ bits still suffice for the extended table D. During a restoration, the search for the correct edge out of a vertex u that is not big can now be performed in $O(g_u) = O(1 + m/(ql)) = O(1 + m/n)$ time. A single restoration carries out the search for q vertices and therefore takes $O((1 + m/n)q) = O((n+m)/\log n)$ time. As in the proof of Theorem 3.1 and for the same reasons, the number of restorations is $O(\log n)$ and the time spent outside of restorations is $O(n+m)$, so the total time for the DFS is $O(n+m)$.

▶ **Lemma 3.2.** *A DFS of a graph with n vertices and m edges, represented via adjacency arrays, can be performed in $O(n+m)$ time with $O(n \log\log n)$ bits.*

3.3 An Upper-Bound Time-Space Tradeoff for DFS

In this section we give a tradeoff between time and space for DFS. Except for the explicit constant factor indicated in the space bound of Theorem 3.1, Theorem 3.3 subsumes Theorem 3.1 and Lemma 3.2, but its proof draws heavily on arguments presented in their proofs.

▶ **Theorem 3.3.** *For every function $t : \mathbb{N} \to \mathbb{N}$ such that $t(n)$ can be computed within the resource bounds of this theorem (e.g., in $O(n)$ time using $O(n)$ bits), a DFS of a graph with n vertices and m edges, represented via adjacency arrays, can be performed in $O((n+m)t(n))$ time with $O(n + n(\log\log n)/t(n))$ bits.*

Proof. Begin by computing a positive integer r with $r = \Theta(1+\frac{\log n}{t(n)})$. Divide S into segments of $q = \Theta(n/\log n)$ consecutive entries each, as usual, but additionally and in the same manner divide S into *big segments* of qr consecutive entries each. Thus each big segment consists of r consecutive usual segments.

We use an algorithm that is a combination of the two algorithms described in Sections 3.1 and 3.2. Its top-level structure is nearly identical to that of the algorithm of Theorem 3.1, except that it employs big segments in place of usual segments. Thus information is maintained about up to two big segments, and occasionally a big segment needs to be restored, which is done in linear time by recoloring all gray vertices white and repeating the computation quietly until the topmost trailer on T has been pushed on S'. The number of such *big restorations* is $O(n/(qr)) = O((\log n)/(1 + \frac{\log n}{t(n)})) = O(t(n))$, so the time spent in big restorations is $O((n+m)t(n))$.

Between big restorations, the algorithm proceeds almost exactly as that of Lemma 3.2. The only differences are that the table D is implemented not as a simple array, but with the dictionary of Lemma 2.1, and that the entry in D of a vertex u is deleted from D when u no longer belongs to one of the two topmost big segments on S. Because of this, the number of bits needed by D is $O(n+qr\log\log n) = O(n+\frac{n\log\log n}{\log n}(1+\frac{\log n}{t(n)})) = O(n+n(\log\log n)/t(n))$. Of course, the restoration should classify a gray vertex without an entry in D as not belonging to the segment under restoration, i.e., such a vertex should not be pushed on S'. The runtime analysis of Section 3.2 carries over to the present context and shows the time spent outside of big restorations to be $O(n+m)$. ◀

4 Reverse DFS with Applications

▶ **Lemma 4.1.** *If we can carry out a DFS $\mathcal{S}$ of a directed graph with n vertices and m edges in time $t(n,m)$ with $s(n,m)$ bits, we can output the reverse of a sequence of symbolic representations of the user calls executed by $\mathcal{S}$ in $O(t(n,m))$ time with $O(s(n,m) + n\log\log n)$ bits.*

Proof. Assume without loss of generality that $n \le s(n,m) \le n\log n$. For times $t_0 < \cdots < t_r$ such that t_0 and t_r are the times of the beginning and the end of $\mathcal{S}$, $r = O(n(\log n)/s(n,m))$, and the number of pushes and pops executed on the stack S of $\mathcal{S}$ during the time interval $I_j = (t_{j-1}, t_j)$ is $O(s(n,m)/\log n)$, for $j = 1,\ldots,r$, use a simulation of an execution of $\mathcal{S}$ to compute r, to label each vertex v with the pair $(d[v], f[v]) \in \{1,\ldots,r\}^2$ such that v becomes gray during $I_{d[v]}$ and black during $I_{f[v]}$, and to record, for $j = 1,\ldots,r$, the top entries H and H' of S at times t_{j-1} and t_j, the value $\widehat{H}$ at time t_{j-1} of the deepest stack entry that changes during I_j, and the first and last user calls, if any, executed during I_j.

For $j = r, \ldots, 1$, we now simulate $\mathcal{S}$ during I_j, record a part of the sequence of (symbolic representations of) user calls executed during I_j, and output the reverse of that sequence, completed with its missing parts, while keeping track of the vertex colors during the corresponding reverse DFS. The missing parts are those calls *preexplore*$(u, v, color[v])$ for which $color[v] \neq white$ and the calls *postexplore*(u, v) that immediately follow them. Without these calls, the sequence of calls executed during I_j fits in $O(s(n, m))$ bits. On the other hand, it is easy to reconstruct the calls missing between two successive recorded calls, essentially by traversing corresponding pieces of adjacency lists or arrays, and to output them in reverse order, $O(n/\log n)$ calls at a time; the details are left to the reader.

To simulate $\mathcal{S}$ from time t_{j-1} onwards, we need the coloring of each vertex v at time t_{j-1}, which can be deduced from $(d[v], f[v])$. We cannot construct the stack valid at time t_{j-1} in its entirety, but since $\mathcal{S}$ is to be simulated only until t_j, it suffices to construct the part of S between $\widehat{H}$ and H, inclusive. We do this by a stack restoration similarly as for DFS: Starting from $\widehat{H}$, each vertex steps through its adjacency list or array, skipping over black and gray outneighbors, until it encounters a first white outneighbor and pushes it on the stack. The recorded stack entries allow us to know exactly when to stop the stack restoration and when to stop the simulation pertaining to I_j. Apart from a constant-factor simulation overhead, the only significant resources needed are $O(s(n, m))$ bits used for the restored stack and for user calls, $O(n \log r) = O(n \log \log n)$ bits to store vertex labels and colors, and the time consumed by the stack reconstructions. For $j = r, \ldots, 1$, if the stack reconstruction prior to the simulation for I_j pushes an entry other than $\widehat{H}$ for a vertex v on S, we must have $f[v] = j$, since otherwise $\widehat{H}$ could not appear at the top of S during I_j. Except for at most r vertices, every vertex is therefore pushed on S in at most one reconstruction, so the time needed for all reconstructions is within a constant factor of the time consumed by $\mathcal{S}$. ◀

In particular, we can output the vertices of a graph G in reverse postorder with respect to a DFS forest of G. If G is directed and acyclic, this order is a topological sorting of G [32].

▶ **Theorem 4.2.** *Within the time and space bounds of a DFS of G, up to a constant factor, plus $O(n \log \log n)$ bits, the vertices of a directed acyclic n-vertex graph G can be output in the order of a topological sorting of G.*

We define the SCC problem as follows: Given a directed n-vertex graph $G = (V, E)$ with c strongly connected components (SCCs), output a sequence $(u_1, k_1), \ldots, (u_n, k_n)$, where $\{u_1, \ldots, u_n\} = V$ and $k_1, \ldots, k_n$ is a nondecreasing sequence of integers such that $1 \leq k_i \leq c$ for $i = 1, \ldots, n$ and $k_i = k_j$, for $1 \leq i, j \leq n$, exactly if u_i and u_j belong to the same SCC of G. Combining Lemma 4.1 with a DFS-based SCC algorithm whose main procedure steps through the vertices in reverse postorder [1], one can easily show the theorem below.

▶ **Theorem 4.3.** *If a DFS of a directed graph with n vertices and m edges, represented with in/out adjacency lists or arrays, can be carried out in $t(n, m)$ time with $s(n, m)$ bits, then, given a directed graph G with n vertices and m edges, represented in the same way, the SCC problem can be solved for G in $O(t(n, m))$ time with $O(s(n, m) + n \log \log n)$ bits.*

5 Computing Connected Components and Breadth-First Search

We consider the following variants of the connected-components and breadth-first search (BFS) problems: The input is an undirected graph $G = (V, E)$ and, in the case of BFS, a permutation $(\pi(1), \ldots, \pi(n))$ of V. The output is a sequence $(u_1, k_1), \ldots, (u_n, k_n)$, where $n = |V|$, $\{u_1, \ldots, u_n\} = V$, and $k_1, \ldots, k_n$ is a nondecreasing sequence of integers with the

following property: For the connected-components problem, $1 \leq k_i \leq c$ for $i = 1, \ldots, n$, where c is the number of connected components of G, and $k_i = k_j$, for $1 \leq i, j \leq n$, exactly if u_i and u_j belong to the same connected component of G. For BFS, k_i is the distance in G from u_i to the first vertex in the sequence $(\pi(1), \ldots, \pi(n))$ that belongs to the same connected component as u_i, for $i = 1, \ldots, n$.

▶ **Theorem 5.1.** *The connected-components and BFS problems for an undirected graph with n vertices and m edges can be solved in $O(n + m)$ time with $O(n)$ bits.*

Proof sketch. For the connected-components problem, we explore the graph using the same (white → gray → black) coloring as in the case of DFS. Instead of exploring an edge incident on the most recently discovered vertex, we pick an arbitrary gray vertex and explore all of its incident edges. When we run out of gray vertices, we instead process a white vertex after incrementing a components counter. If the set of gray vertices is stored in an instance of the dictionary of Lemma 2.1 with its *some_id* operation, the process can easily be carried out in linear time.

For the BFS problem, we refine the process by splitting the set of gray vertices in two, the sets of *inner-gray* and of *outer-gray* vertices. As long as there are inner-gray vertices, we process one of these, coloring its white neighbors outer-gray. When this is no longer the case, we increment a distance counter and recolor the outer-gray vertices inner-gray. When there are neither inner-gray nor outer-gray vertices, we set the distance counter to 0 and continue the process at the first white vertex in the sequence $(\pi(1), \ldots, \pi(n))$. ◀

6 Priority Queues with a Deletion Budget and Their Applications

For our purposes, a *priority queue* is a data structure that maintains an initially empty collection of items, each with a unique *identification*, a *key* drawn from a totally ordered set, and arbitrary *satellite data*, under the operations *insert*, *extract_min* and *decrease_key*. The operation *insert* inserts a new item in the collection, *extract_min* returns an item whose key is minimal after deleting it from the collection, and a call *decrease_key*(v, d, p) replaces the current key d_v of the item with identification v by d and its current satellite data by p, provided that $d < d_v$, and does nothing if $d \geq d_v$. Our priority queue is nonstandard in two minor ways. First, satellite data are frequently not included in the specification of priority queues. And second, a call *decrease_key*(v, d, p) is usually considered legal only if d is smaller than the key of v before the call.

We will say that a priority queue has a *deletion budget* of b if it is guaranteed to work correctly until the end of the bth call of *extract_min* (but possibly not after that). Thus usual priority queues have infinite deletion budgets. The following general construction derives from a priority queue Q a priority queue Q_b with a smaller budget b: Initially, Q_b operates exactly as Q. Whenever the number of items stored in Q_b reaches $2b$, however, all the items are extracted from Q_b, their median is computed, b items with largest keys are thrown away, and the other b items are reinserted in Q_b. A call of *decrease_key* that refers to an item that was thrown away reinserts the item with its new key. To see the correctness of the construction, observe that if an item is thrown away and not later reinserted with a smaller key, Q can avoid returning it in one of the b first calls of *extract_min*, whereas an item whose key in Q_b is incorrect and therefore too large (the item must have been thrown away and later reinserted) is definitely not returned by Q_b in any of these calls. The main advantage of a priority queue with a small deletion budget is that it uses little space. By applying the construction above to a Fibonacci heap [25], augmented with an instance of

the dictionary of Lemma 2.1 that we use to map identifications of items to their positions in the Fibonacci heap and their keys and satellite data, we obtain:

▶ **Lemma 6.1.** *For every given $n, b \in \mathbb{N}$, there is a priority queue with deletion budget b for identifications drawn from $\{1, \ldots, n\}$ with $O(\log n)$-bit keys and $O(\log n)$-bit satellite data that executes insert and decrease_key in constant amortized time and extract_min in $O(\log n)$ amortized time and that uses $O(n + b \log n)$ bits.*

6.1 Computing Minimum Spanning Forests

▶ **Theorem 6.2.** *Given an undirected graph G with n vertices, m edges and $O(\log n)$-bit edge weights, represented with adjacency arrays and cross pointers, the edges of a minimum spanning forest of G can be output either in $O(n + m \log n)$ time with $O(n)$ bits or in $O(m + n \log n)$ time with $O(n \log(2 + m/n))$ bits.*

Proof sketch. We give a proof only for the more interesting case $m \geq n/2$. We run Prim's algorithm [30] with an instance Q_b of the priority queue of Lemma 6.1 with deletion budget $\Theta(n/\log n)$. Prim's algorithm grows a minimum spanning forest F one tree at a time, repeatedly adding to F a vertex outside of F that is closest to F. For each vertex v outside of F, Q_b stores the item (v, d_v, p_v), where the key d_v is the smallest weight of an edge $\{u, v\}$ for which u is in F, and p_v, if $d_v < \infty$, is the position of u in the adjacency array $N(v)$ of v.

Q_b must be refilled $O(\log n)$ times. Between the refillings, the algorithm n times executes an *extract_min* operation on Q_b to obtain an item (v, d_v, p_v) and, if $d_v < \infty$, outputs the edge $\{u, v\}$, where u is the vertex in position p_v in $N(v)$. For each neighbor x of v outside of F, it also executes the operation *decrease_key*(x, d, p), where d is the weight of the edge $\{v, x\}$ and p is the position of v in $N(x)$—which can be found by following a cross pointer. Outside of refillings of Q_b, the algorithm uses $O(m + n \log n)$ time.

Processing every edge in G, we can refill Q_b in $O(m)$ time, which shows the first part of the theorem. In order to be faster, we maintain for each vertex v outside of F the position in its adjacency array of a closest neighbor of v in F, i.e., the last component of the triple (v, d_v, p_v). This needs $O(n \log(2 + m/n))$ bits and allows the refilling time to be lowered to $O(n)$, which shows the second part of the theorem. ◀

6.2 The Single-Source Shortest-Paths Problem

▶ **Theorem 6.3.** *For every function $s : \mathbb{N} \to \mathbb{N}$ such that $s(n)$ can be computed within the resource bounds of this theorem (e.g., in $O(n)$ time using $O(n)$ bits), the following problem can be solved in $O(m + n \log n + n((\log n)/s(n))^2)$ time with $O(n(\log(2 + m/n) + s(n)))$ bits: Given a directed graph $G = (V, E)$ with n vertices, m edges and nonnegative $O(\log n)$-bit edge weights, represented with sorted in/out adjacency arrays and cross pointers, and a vertex $s^* \in V$ from which all vertices in G are reachable, compute a shortest-paths tree in G rooted at s^*, i.e., a tree that is the union, over all $v \in V$, of a shortest path in G from s^* to v.*

Proof sketch. Assume without loss of generality that $s(n) \leq \log n$. We run Dijkstra's algorithm [15, 19, 34] with an instance Q_b of the priority queue of Lemma 6.1 with deletion budget $\Theta(ns(n)/\log n)$. Dijkstra's algorithm grows an SSSP tree T rooted at s^* one vertex at a time, repeatedly adding to T a vertex outside of T that is closest to s^*. For each vertex v outside of T, Q_b stores the item (v, d_v, p_v), where the key d_v is the infimum of the lengths of paths in G from s^* to v whose only vertex outside of T is v, and p_v, if $d_v < \infty$, is the position in the in-adjacency array $N_{\text{in}}(v)$ of v of the second-last vertex on a shortest such path.

Q_b must be refilled $O(\log n/s(n))$ times. Between the refillings, the algorithm n times executes an *extract_min* operation on Q_b to obtain an item (v, d_v, p_v). It adds v to T and, if $v \neq s^*$, adds also the edge (u, v), where u is the vertex in position p_v in $N_{\text{in}}(v)$. The edge (u, v) or the distance d_v from s^* to v may be output at this point; at any rate, the algorithm remembers (u, v) by storing p_v permanently with v. This allows v to find its parent u in T in constant time—an operation that we call *following a parent pointer*—and, over all vertices v, needs $O(n \log(2 + m/n))$ bits. For each outneighbor x of v outside of T, the algorithm also executes the operation *decrease_key*$(x, d_v + c, p)$, where c is the weight of the edge (v, x) and p is the position of v in $N_{\text{in}}(x)$, an operation known as *relaxing* the edge (v, x). Outside of refillings of Q_b, the algorithm uses $O(m + n \log n)$ time.

To ease refillings, the algorithm maintains the following additional information: First, a list L of the vertices added to T since the previous refilling and their distances from s^* ($O(ns(n))$ bits). Second, for each vertex v outside of T, the integer p_v such that a triple of the form (v, d, p_v) was present in Q_b at the end of the previous refilling ($O(n \log(2 + m/n))$ bits). We call p_v the *old tentative parent pointer* of v.

In each refilling, the vertices outside of T are processed in $O(\log n/s(n))$ *batches* of $O(r)$ vertices each, where $r = ns(n)/\log n$. The batches must be consistent with the ordering of the adjacency arrays of G in the sense that if u and v are vertices such that v appears in a batch after that of u, v may not precede u in any adjacency array. The purpose of the processing of a batch is, for each vertex v in the batch, to insert the correct triple (v, d_v, p_v) in Q_b and to store p_v as the new old tentative parent pointer of v. We will show that a batch can be processed in $O(n)$ time plus a quantity that sums to $O(m + n \log n)$ over all refillings. Since there are $O(\log n/s(n))$ refillings and $O(\log n/s(n))$ batches to be processed in each refilling, we arrive at the overall time bound of $O(m + n \log n + n((\log n)/s(n))^2)$.

For each batch, we first recompute the items stored in Q_b for the vertices in the batch at the end of the previous refilling of Q_b. Since each vertex v in the batch already knows its old tentative parent pointer p_v, this amounts to computing the length of the path in T from s^* to the vertex u in position p_v in $N_{\text{in}}(v)$. This quantity could be found simply by following parent pointers from u to s^*, summing edge weights along the way. In the interest of efficiency, however, the vertices in the batch collaborate.

In a first phase, the vertices in the batch, one by one, emit a token that follows parent pointers, summing edge weights as it goes along, but marks the vertices that it passes and stops as soon as it reaches s^* or a vertex marked earlier by another token, after marking that vertex as a *branching vertex*. The total number of edges on the paths traversed by the tokens is bounded by $n - 1$ (T has no more edges), and the number of branching vertices is $O(r)$.

In a second phase, the vertices are processed in the same order as in the first phase, and each sends a token twice along the same path as in the first phase. The path ends either at s^*, at a distance of 0 from itself, or at a branching vertex which, at this point, will have been marked with its distance from s^*. Adding that distance to the known length of the path traversed, the vertex obtains its own distance from s^*. In the final traversal of the path by its token, the vertex helps later vertices in the batch by marking all branching vertices along the path with their distances from s^*. This is done by subtracting the edge weights encountered along the path from a variable initialized with the total length of the path.

What remains for the batch at this point is to relax all edges from vertices stored in L to vertices in the batch. Because the adjacency arrays are sorted consistently with the batches, this can be done in $O(m+n \log n)$ time over all refillings, which establishes the running time anticipated above. If the distances of branching vertices from s^* are stored in an instance of the dictionary of Lemma 2.1, the necessary additional space is $O(n+r \log n) = O(ns(n))$. ◀

References

1 Alfred V. Aho, John E. Hopcroft, and Jeffrey D. Ullman. *Data Structures and Algorithms*. Addison-Wesley, 1983.

2 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.

3 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.

4 Tetsuo Asano, Wolfgang Mulzer, Günter Rote, and Yajun Wang. Constant-work-space algorithms for geometric problems. *J. Comput. Geom.*, 2(1):46–68, 2011.

5 Luis Barba, Matias Korman, Stefan Langerman, and Rodrigo I. Silveira. Computing a visibility polygon using few variables. *Comput. Geom. Theory Appl.*, 47(9):918–926, 2014.

6 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.

7 Greg Barnes, Jonathan F. Buss, Walter L. Ruzzo, and Baruch Schieber. A sublinear space, polynomial time algorithm for directed *s-t* connectivity. *SIAM J. Comput.*, 27(5):1273–1282, 1998.

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

9 Hervé Brönnimann, John Iacono, Jyrki Katajainen, Pat Morin, Jason Morrison, and Godfried Toussaint. Space-efficient planar convex hull algorithms. *Theor. Comput. Sci.*, 321(1):25–40, 2004.

10 Timothy M. Chan. Comparison-based time-space lower bounds for selection. *ACM Trans. Algorithms*, 6(2):Article 26, 2010.

11 Timothy M. Chan and Eric Y. Chen. Multi-pass geometric algorithms. *Discrete Comput. Geom.*, 37(1):79–102, 2007.

12 Timothy M. Chan, J. Ian Munro, and Venkatesh Raman. Faster, space-efficient selection algorithms in read-only memory for integers. In *Proc. 24th International Symposium on Algorithms and Computation (ISAAC 2013)*, volume 8283 of *LNCS*, pages 405–412. Springer, 2013.

13 Timothy M. Chan, J. Ian Munro, and Venkatesh Raman. Selection and sorting in the "restore" model. In *Proc. 25th Annual ACM-SIAM Symposium on Discrete Algorithms (SODA 2014)*, pages 995–1004. SIAM, 2014.

14 Thomas H. Cormen, Charles E. Leiserson, Ronald L. Rivest, and Clifford Stein. *Introduction to Algorithms*. The MIT Press, 3rd edition, 2009.

15 George B. Dantzig. On the shortest route through a network. *Manag. Sci.*, 6(2):187–190, 1960.

16 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.

17 Bireswar Das, Samir Datta, and Prajakta Nimbhorkar. Log-space algorithms for paths and matchings in k-trees. *Theory Comput. Syst.*, 53(4):669–689, 2013.

18 Bireswar Das, Jacobo Torán, and Fabian Wagner. Restricted space algorithms for isomorphism on bounded treewidth graphs. *Inform. Comput.*, 217:71–83, 2012.

19 E. W. Dijkstra. A note on two problems in connexion with graphs. *Numer. Math.*, 1(1):269–271, 1959.

20 Jeff Edmonds, Chung Keung Poon, and Dimitris Achlioptas. Tight lower bounds for *st*-connectivity on the NNJAG model. *SIAM J. Comput.*, 28(6):2257–2284, 1999.

21 Michael Elberfeld, Andreas Jakoby, and Till Tantau. Logspace versions of the theorems of Bodlaender and Courcelle. In *Proc. 51st Annual IEEE Symposium on Foundations of Computer Science (FOCS 2010)*, pages 143–152. IEEE Computer Society, 2010.

22 Michael Elberfeld and Ken-ichi Kawarabayashi. Embedding and canonizing graphs of bounded genus in logspace. In *Proc. 46th ACM Symposium on Theory of Computing (STOC 2014)*, pages 383–392. ACM, 2014.

23 Amr Elmasry, Daniel Dahl Juhl, Jyrki Katajainen, and Srinivasa Rao Satti. Selection from read-only memory with limited workspace. *Theor. Comput. Sci.*, 554:64–73, 2014.

24 Greg N. Frederickson. Upper bounds for time-space trade-offs in sorting and selection. *J. Comput. Syst. Sci.*, 34(1):19–26, 1987.

25 Michael L. Fredman and Robert Endre Tarjan. Fibonacci heaps and their uses in improved network optimization algorithms. *J. ACM*, 34(3):596–615, 1987.

26 Torben Hagerup and Frank Kammer. Dynamic data structures for the succinct RAM, 2015. In preparation.

27 J. I. Munro and M. S. Paterson. Selection and sorting with limited storage. *Theor. Comput. Sci.*, 12(3):315–323, 1980.

28 J. Ian Munro and Venkatesh Raman. Selection from read-only memory and sorting with minimum data movement. *Theor. Comput. Sci.*, 165(2):311–323, 1996.

29 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.

30 R. C. Prim. Shortest connection networks and some generalizations. *Bell Syst. Tech. J.*, 36(6):1389–1401, 1957.

31 Venkatesh Raman and Sarnath Ramnath. Improved upper bounds for time-space trade-offs for selection. *Nord. J. Comput.*, 6(2):162–180, 1999.

32 Robert Tarjan. Finding dominators in directed graphs. *SIAM J. Comput.*, 3(1):62–89, 1974.

33 Martin Tompa. Two familiar transitive closure algorithms which admit no polynomial time, sublinear space implementations. *SIAM J. Comput.*, 11(1):130–137, 1982.

34 P. D. Whiting and J. A. Hillier. A method for finding the shortest route through a road network. *J. Oper. Res. Soc.*, 11(1/2):37–40, 1960.