A Framework for In-place Graph Algorithms

Sankardeep Chakraborty[1]
The University of Tokyo, Japan
sankardeep@me2.mist.i.u-tokyo.ac.jp

Anish Mukherjee
Chennai Mathematical Institute, India
anish@cmi.ac.in

Venkatesh Raman
The Institute of Mathematical Sciences, HBNI, India
vraman@imsc.res.in

Srinivasa Rao Satti
Seoul National University, South Korea
ssrao@cse.snu.ac.kr

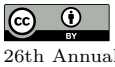
Abstract

Read-only memory (ROM) model is a classical model of computation to study time-space tradeoffs of algorithms. A classical result on the ROM model is that any algorithm to sort n numbers using $O(s)$ words of extra space requires $\Omega(n^2/s)$ comparisons for $\lg n \leq s \leq n/\lg n$[2] and the bound has also been recently matched by an algorithm. However, if we relax the model, we do have sorting algorithms (say Heapsort) that can sort using $O(n \lg n)$ comparisons using $O(\lg n)$ bits of extra space, even keeping a permutation of the given input sequence at anytime during the algorithm.

We address similar relaxations for graph algorithms. We show that a simple natural relaxation of ROM model allows us to implement fundamental graph search methods like BFS and DFS more space efficiently than in ROM. By simply allowing elements in the adjacency list of a vertex to be permuted, we show that, on an undirected or directed connected graph G having n vertices and m edges, the vertices of G can be output in a DFS or BFS order using $O(\lg n)$ bits of extra space and $O(n^3 \lg n)$ time. Thus we obtain similar bounds for *reachability* and *shortest path distance* (both for undirected and directed graphs). With a little more (but still polynomial) time, we can also output vertices in the *lex-DFS* order. As reachability in directed graphs (even in DAGs) and shortest path distance (even in undirected graphs) are NL-complete, and lex-DFS is P-complete, our results show that our model is more powerful than ROM if $\mathsf{L} \neq \mathsf{P}$.

En route, we also introduce and develop algorithms for another relaxation of ROM where the adjacency lists of the vertices are circular lists and we can modify only the heads of the lists. Here we first show a linear time DFS implementation using $n + O(\lg n)$ bits of extra space. Improving the extra space exponentially to only $O(\lg n)$ bits, we also obtain BFS and DFS albeit with a slightly slower running time. Both the models we propose maintain the graph structure throughout the algorithm, only the order of vertices in the adjacency list changes. In sharp contrast, for BFS and DFS, to the best of our knowledge, there are no algorithms in ROM that use even $O(n^{1-\epsilon})$ bits of extra space; in fact, implementing DFS using cn bits for $c < 1$ has been mentioned as an open problem. Furthermore, DFS (BFS, respectively) algorithms using $n + o(n)$ ($o(n)$, respectively) bits of extra use Reingold's [JACM, 2008] or Barnes et al's reachability algorithm [SICOMP, 1998] and hence have high runtime. Our results can be contrasted with the recent result of Buhrman et al. [STOC, 2014] which gives an algorithm for *directed st-reachability* on *catalytic Turing machines* using $O(\lg n)$ bits with catalytic space $O(n^2 \lg n)$ and time $O(n^9)$.

[1] This work was partially supported by JST CREST Grant Number JPMJCR1402, Japan.
[2] We use lg to denote logarithm to the base 2.


© Sankardeep Chakraborty, Anish Mukherjee, Venkatesh Raman, and Srinivasa Rao Satti;
licensed under Creative Commons License CC-BY
26th Annual European Symposium on Algorithms (ESA 2018).
Editors: Yossi Azar, Hannah Bast, and Grzegorz Herman; Article No. 13; pp. 13:1–13:16

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

2012 ACM Subject Classification Mathematics of computing → Graph algorithms, Theory of computation → Models of computation

Keywords and phrases DFS, BFS, in-place algorithm, space-efficient graph algorithms, logspace

Digital Object Identifier 10.4230/LIPIcs.ESA.2018.13

Related Version A full version of the paper can be found at [21], `https://arxiv.org/abs/1711.09859`.

1 Introduction

Motivated by the rapid growth of huge data set ("big data"), space efficient algorithms are becoming increasingly important than ever before. The proliferation of specialized handheld devices and embedded systems that have a limited supply of memory provide another motivation to consider space efficient algorithms. To design *space-efficient* algorithms in general, several models of computation have been proposed. Among them, the following two computational models have received considerable attention in the literature.

- In the *read-only* memory (ROM) model, we assume that the input is given in a read-only memory. The output of an algorithm is written on a separate write-only memory, and the output can not be read or modified again. In addition to the input and output media, a limited random access workspace is available. Early work on this model was on designing lower bounds [14, 15, 16], for designing algorithms for selection and sorting [26, 38, 43, 49, 50, 52] and problems in computational geometry [6, 9, 12, 25, 30]. Recently there has been interest on space-efficient graph algorithms [7, 10, 11, 22, 23, 24, 37, 44, 45].
- In the *in-place* model, the input elements are given in an array, and the algorithm may use the input array as working space. Hence the algorithm may modify the array during its execution. After the execution, all the input elements should be present in the array (maybe in a permuted order) and the output maybe put in the same array or sent to an output stream. The extra space usage during the execution of the algorithm is limited to $O(\lg n)$ bits. A prominent example of an in-place algorithm is the classic heap-sort. Other than in-place sorting [42], searching [40, 51] and selection [47], many in-place algorithms were designed in areas such as computational geometry [17] and stringology [41].

Apart from these models, researchers have also considered (semi)-streaming models [3, 39, 49] for designing space-efficient algorithms. Very recently the following two new models were introduced in the literature with the same objective.

- Chan et al. [27] introduced the *restore* model which is a more relaxed version of read-only memory (and a restricted version of the in-place model), where the input is allowed to be modified, but at the end of the computation, the input has to be restored to its original form. They also gave space efficient algorithms for selection and sorting on integer arrays in this model. This has motivation, for example, in scenarios where the input (in its original form) is required by some other application.
- Buhrman et al. [18, 19, 46] introduced and studied the *catalytic-space* model where a small amount (typically $O(\lg n)$ bits) of clean space is provided along with some large additional auxiliary space, with the condition that the additional space is initially in an arbitrary, possibly incompressible, state and must be returned to this state when the computation is finished. The input is assumed to be given in ROM. Thus this model can be thought of as having an auxiliary storage that needs to be 'restored' in contrast to the model by Chan et al. [27] where the input array has to be 'restored'. They show various

interesting complexity theoretic consequences in this model and designed space-efficient algorithms in comparison with the ROM model for a few combinatorial problems.

1.1 Previous work on space efficient graph algorithms

Even though these models were introduced in the literature with the aim of designing and/or implementing various algorithms space efficiently, *space efficient graph algorithms* have been designed only in the (semi)-streaming and the ROM model. In the streaming and semi-streaming models, researchers have studied several basic and fundamental algorithmic problems such as connectivity, minimum spanning tree, matching. See [48] for a comprehensive survey in this field. Research on these two models (i.e., streaming and semi-streaming) is relatively new and has been going on for the last decade or so whereas the study in ROM could be traced back to almost 40 years. In fact there is already a rich history of designing space efficient algorithms in the read-only memory model. The complexity class L is the class containing decision problems that can be solved by a deterministic Turing machine using only logarithmic amount of work space for computation. There are several important algorithmic results [31, 34, 35, 36] for this class, the most celebrated being Reingold's method [55] for checking *st*-reachability in an undirected graph, i.e., to determine if there is a path between two given vertices s and t. NL is the non-deterministic analogue of L and it is known that the *st*-reachability problem for *directed* graphs is NL-complete (with respect to log space reductions). Using Savitch's algorithm [5], this problem can be solved in $n^{O(\lg n)}$ time using $O(\lg^2 n)$ bits of extra space. Savitch's algorithm is very space efficient but its running time is superpolynomial. Among the deterministic algorithms running in polynomial time for directed *st*-reachability, the most space efficient algorithm is due to Barnes et al. [13] who gave a slightly sublinear space (using $n/2^{\Theta(\sqrt{\lg n})}$ bits) algorithm for this problem running in polynomial time. We know of no better polynomial time algorithm for this problem with better space bound. Moreover, the space used by this algorithm matches a lower bound on space for solving directed *st*-reachability on a restricted model of computation called Node Naming Jumping Automata on Graphs (NNJAG's) [28, 33]. This model was introduced especially for the study of directed *st*-reachability and most of the known sublinear space algorithms for this problem can be implemented on it. Thus, to design any polynomial time ROM algorithm taking space less than $n/2^{\Theta(\sqrt{\lg n})}$ bits requires radically new ideas. Recently there has been some improvement in the space bound for some special classes of graphs like planar and H-minor free graphs [8, 20]. A drawback for all these algorithms using small space i.e., sublinear number of bits, is that their running time is often some polynomial of high degree. This is not surprising as Tompa [57] showed that for directed *st*-reachability, if the number of bits available is $o(n)$ then some natural algorithmic approaches to the problem require super-polynomial time.

Motivated by these impossibility results from complexity theory and inspired by the practical applications of these fundamental graph algorithms, recently there has been a surge of interest in improving the space complexity of the fundamental graph algorithms without paying too much penalty in the running time i.e., reducing the working space of the classical graph algorithms to $O(n)$ bits with little or no penalty in running time. Thus the goal is to design space-efficient yet reasonably time-efficient graph algorithms on the ROM. Generally most of the classical linear time graph algorithms take $O(n \lg n)$ bits. Recently Asano et al. [7] gave an $O(m \lg n)$ time algorithm using $O(n)$ bits, and another implementation taking $n + o(n)$ bits (using Reingold's or Barnes et al's reachability algorithm) but using high polynomial running time. Later, time bound was improved to $O(m \lg \lg n)$ still using $O(n)$

bits in [37]. For sparse graphs, the time bound is further improved in [10, 22] to optimal $O(m)$ using still $O(n)$ bits of space. Improving on the classical linear time implementation of BFS which uses $O(n \lg n)$ bits of space, recent space efficient algorithms [10, 37, 44] have resulted in a linear time algorithm using $n \lg 3 + o(n)$ bits. We know of no algorithm for BFS using $n + o(n)$ bits and $O(m \lg^c n)$ (or even $O(mn)$) time for some constant c in ROM. The only BFS algorithm taking sublinear space uses $n/2^{\Theta(\sqrt{\lg n})}$ bits [13] and has a high polynomial runtime. A few other space efficient algorithms for fundamental graph problems like checking strong connectivity [37], biconnectivity and performing *st*-numbering [22], recognizing chordal and outerplanar graphs [24, 45] were also designed very recently.

1.2 In-place model for graph algorithms

In order to break these inherent space bound barriers and obtain reasonably time and space efficient graph algorithms, we want to relax the limitations of ROM. And the most natural and obvious candidate in this regard is the classical *in-place* model. Thus our main objective is to initiate a systematic study of efficient in-place (i.e., using $O(\lg n)$ bits of extra space) algorithms for graph problems. To the best of our knowledge, this has not been done in the literature before. Our first goal towards this is to properly define models for in-place graph algorithms. As in the case of the standard in-place model, we need to ensure that the graph (adjacency) structure remains intact throughout the algorithm. Let $G = (V, E)$ be the input graph with $n = |V|$, $m = |E|$, and assume that the vertex set V of G is the set $V = \{1, 2, \cdots, n\}$. To describe these models, we assume that the input graph representation consists of two parts: (i) an array V of length n, where $V[i]$ stores a pointer to the adjacency list of vertex i, and (ii) a list of singly linked lists, where the i-th list consists of a singly linked list containing all the neighbors of vertex i with $V[i]$ pointing to the head of the list. In the ROM model, we assume that both these components cannot be modified. In our relaxed models, we assume that one of these components can be modified in a limited way. This gives rise to two different models which we define next.

Implicit model. The most natural analogue of in-place model allows any two elements in the adjacency list of a vertex to be swapped (in $O(1)$ time assuming that we have access to the nodes storing those elements in the singly linked list). The adjacency "structure" of the representation does not change; only the values stored can be swapped. (One may restrict this further to allow only elements in adjacent nodes to be swapped. Most of our algorithms work with this restriction.) We call it the implicit model inspired by the notion of *implicit data structures* [51].

Rotate model. In this model, we assume that only the pointers stored in the array V can be modified, that too in a limited way - to point to any node in the adjacency list, instead of always pointing to the first node. In space-efficient setting, since we do not have additional space to store a pointer to the beginning of the adjacency list explicitly, we assume that the second component of the graph representation consists of a list of circular linked lists (instead of singly linked lists) – i.e., the last node in the adjacency list of each vertex points to the first node (instead of storing a null pointer). We call the element pointed to by the pointer as the front of the list, and a unit cost rotate operation changes the element pointed to by the pointer to the next element in the list.

Thus the rotate model corresponds to keeping the adjacency lists in read-only memory and allowing (limited) updates on the pointer array that points to these lists. And, the implicit model corresponds to the reverse case, where we keep the pointer array in read-only

memory and allow swaps on the adjacency lists/arrays. A third alternative especially for the implicit model is to assume that the input graph is represented as an adjacency array, i.e., adjacency lists are stored as arrays instead of singly linked lists (see [22, 37, 45] for some results using this model); and we allow here that any two elements in the adjacency array can be swapped. In this model, some of our algorithms have improved performance in time.

1.3 Definitions, computational complexity and notations

We study some basic and fundamental graph problems in these models. In what follows we provide the definitions and state the computational complexity of these problems. For the DFS problem, there have been two versions studied in the literature. In the *lexicographically smallest DFS* or *lex-DFS* problem, when DFS looks for an unvisited vertex to visit in an adjacency list, it picks the "first" unvisited vertex where the "first" is with respect to the appearance order in the adjacency list. The resulting DFS tree will be unique. In contrast to lex-DFS, an algorithm that outputs *some* DFS numbering of a given graph, treats an adjacency list as a set, ignoring the order of appearance of vertices in it, and outputs a vertex ordering T such that there exists *some* adjacency ordering R such that T is the DFS numbering with respect to R. We say that such a DFS algorithm performs *general-DFS*. Reif [54] has shown that lex-DFS is P-complete (with respect to log-space reductions) implying that a logspace algorithm for lex-DFS results in the collapse of complexity classes P and L. Anderson et al. [4] have shown that even computing the leftmost root-to-leaf path of the lex-DFS tree is P-complete. For many years, these results seemed to imply that the general-DFS problem, that is, the computation of any DFS tree is also inherently sequential. However, Aggarwal et al. [1, 2] proved that the general-DFS problem can be solved much more efficiently, and it is in RNC. Whether the general-DFS problem is in NC is still open.

As is standard in the design of space-efficient algorithms [10, 37], while working with directed graphs, we assume that the graphs are given as in/out (circular) adjacency lists i.e., for a vertex v, we have the (circular) lists of both in-neighbors and out-neighbors of v. We assume the word RAM model of computation where the machine consists of words of size w in $\Omega(\lg n)$ bits and any logical, arithmetic and bitwise operation involving a constant number of words takes $O(1)$ time. We count space in terms of number of *extra* bits used by the algorithm other than the input, and this quantity is referred as "extra space" and "space" interchangeably throughout the paper. By a path of length d, we mean a simple path on d edges. By $deg(x)$ we mean the degree of the vertex x. In directed graphs, it should be clear from the context whether that denotes out-degree or in-degree. By a BFS/DFS traversal of the input graph G, as in [7, 10, 22, 37, 44], we refer to reporting the vertices of G in the BFS/DFS ordering, i.e., in the order in which the vertices are visited for the first time.

1.4 Our Results

Depth-first Search. In the rotate model, we show the following in Sections 2.1, 2.2 and 2.3.

▶ **Theorem 1.** *Let G be a directed or an undirected graph, and $\ell \leq n$ be the maximum depth of the DFS tree starting at a source vertex s. Then in the* rotate *model, the vertices of G can be output in*

(a) *the lex-DFS order in $O(m+n)$ time using $n \lg 3 + O(\lg^2 n)$ bits of extra space,*
(b) *a general-DFS order in $O(m+n)$ time using $n + O(\lg n)$ bits of extra space, and*
(c) *a general-DFS order in $O(m^2/n + m\ell)$ time for an undirected graph and in $O(m(n+\ell^2))$ time for directed graphs using $O(\lg n)$ bits of extra space. For this algorithm, we assume that s can reach all other vertices.*

In the implicit model, we obtain polynomial time implementations for lex-DFS and general-DFS using $O(\lg n)$ bits of extra space. For lex-DFS, this is conjectured to be unlikely in ROM as the problem is P-complete [54]. In particular, we show the following in Section 4.

▶ **Theorem 2.** *Let G be a directed or an undirected graph with a source vertex s and $\ell \leq n$ be the maximum depth of any DFS tree starting at s that can reach all other vertices. Then in the* implicit *model, using $O(\lg n)$ bits of extra space, the vertices of G can be output in*

(a) *the lex-DFS order in $O(m^3/n^2 + \ell m^2/n)$ time if G is given in adjacency list and in $O(m^2 \lg n/n)$ time if G is given in adjacency array for undirected graphs. For directed graphs our algorithm takes $O(m^2(n + \ell^2)/n)$ time if G is given in adjacency list and $O(m \lg n(n + \ell^2))$ time if G is given in adjacency array;*

(b) *a general-DFS traversal order in $O(m^2/n)$ time if the input graph G is given in an adjacency list and in $O(m^2(\lg n)/n + m\ell \lg n))$ time if it is given in an adjacency array.*

In sharp contrast, for space efficient algorithms for DFS in ROM, the landscape looks markedly different. To the best of our knowledge, there are no DFS algorithms in general graphs in ROM that use $O(n^{1-\epsilon})$ bits. In fact, an implementation of DFS taking cn bits for $c < 1$ has been proposed as an open problem in [7].

Breadth-first Search. In the rotate model, we show the following.

▶ **Theorem 3.** (♠)[3] *Let G be a directed or an undirected graph, and $\ell \leq n$ be the depth of the BFS tree starting at the source vertex s. Then in the* rotate *model, the vertices of G can be output in a BFS order in*

(a) *$O(m + n\ell^2)$ time using $n + O(\lg n)$ bits of extra space, and*

(b) *$O(m\ell + n\ell^2)$ time using $O(\lg n)$ bits of extra space. Here we assume that the source vertex can reach all other vertices.*

In the implicit model, we can match the runtime of BFS from rotate model, and do better in some special classes of graphs. In particular, we show the following.

▶ **Theorem 4.** (♠) *Let G be a directed or an undirected graph with a source vertex that can reach all other vertices by a distance of at most $\ell \leq n$. Then in the* implicit *model, using $O(\lg n)$ bits of extra space, the vertices of G can be output in a BFS order in*

(a) *$O(m + n\ell^2)$ time;*

(b) *the runtime can be improved to $O(m + n\ell)$ time if there are no degree 2 vertices;*

(c) *the runtime can be improved to $O(m)$ if the degree of every vertex is at least $2 \lg n + 3$.*

Similar to DFS, to the best of our knowledge, there are no polynomial time BFS algorithms in ROM that use even $O(n^{1-\epsilon})$ bits. On the other hand, we don't hope to have a BFS algorithm (for both undirected and directed graphs) using $O(\lg n)$ bits of extra space in ROM as the problem is NL-complete [5].

Minimum Spanning Tree (MST). We also study the problem of reporting the edges of a MST of a given undirected connected graph G and we show the following.

[3] Proofs of results marked with (♠) appear in the full version [21].

▶ **Theorem 5.** (♠) *A minimum spanning tree of a given undirected weighted graph G can be found using $O(\lg n)$ bits of extra space and in*

(a) *$O(mn)$ time in the* rotate *model,*

(b) *$O(mn^2)$ time in the* implicit *model if G is given in an adjacency list, and*

(c) *$O(mn \lg n)$ time in the* implicit *model when G is represented in an adjacency array.*

Note that by the results of [53, 55], we already know log-space algorithms for MST in ROM but again the drawback of those algorithms is their large time complexity. On the other hand, our algorithms have relatively small polynomial running time, simplicity, making it an appealing choice in applications with strict space constraints.

1.5 Techniques

Our implementations follow (variations of) the classical algorithms for BFS and DFS that use three colors (white, gray and black), but avoid the use of stack (for DFS) and queue (for BFS). In the rotate model, we first observe that in the usual search algorithms one can dispense with the extra data structure space of pointers maintaining the search tree (while retaining the linear number of bits and a single bit per vertex in place of the full unvisited/visited/explored array) simply by rotating each circular adjacency lists to move the parent or a (typically the currently explored) child to the beginning of the list to help navigate through the tree during the forward or the backtracking step, i.e. by changing the pointer from the vertex to the list of its adjacencies by one node at a time. This retains the basic efficiency of the search strategies. The nice part of this strategy is that the total number of rotations also can be bounded. To reduce the extra space from linear to logarithmic, it is noted that one can follow the vertices based on the current rotations at each vertex to determine the visited status of a vertex, i.e. these algorithms use the rotate operation in a non-trivial way to move elements within the lists to determine the color of the vertices as well. However, the drawback is that to do so could require moving up (or down) the full height of the implicit search tree. This yields polynomial rather than (near-)linear time algorithms.

In the implicit model, we use the classical *bit encoding* trick used in the development of the implicit data structures [51]. We encode one (or two) bit(s) using a sequence of two (or three respectively) distinct numbers. To encode a single bit b using two distinct values x and y with $x < y$, we store the sequence x, y if $b = 0$, and y, x otherwise. Similarly, permuting three distinct values x, y, z with $x < y < z$, we can represent six combinations. We can choose any of the four combinations to represent up to 4 colors (i.e. two bits). Generalizing this further, we can encode a pointer taking $\lg n$ bits using $2 \lg n$ distinct elements where reading or updating a bit takes constant time, and reading or updating a pointer takes $O(\lg n)$ time. This also is the reason for the requirement of vertices with degree at least 3 or $2 \lg n + 3$ for faster algorithms, which will become clear in the description of the algorithms.

1.6 Consequences of our BFS and DFS results

There are many interesting and surprising consequences of our results for BFS and DFS in both the rotate and implicit model. In what follows, we mention a few of them. See the full version [21] for the complete discussion on the consequences of our results.

- For *directed st-reachability*, as mentioned previously, the most space efficient polynomial time algorithm [13] uses $n/2^{\Theta(\sqrt{\lg n})}$ bits. In sharp contrast, we obtain efficient (timewise) log-space algorithms for this problem in both the rotate and implicit models (as a corollary of our directed graph DFS/BFS results). In terms of workspace this is exponentially

better than the best known polynomial time algorithm [13] for this problem in ROM. For us, this provides one of the main motivations to study this model. A somewhat incomparable result obtained recently by Buhrman et al. [18, 46] where they designed an algorithm for *directed st-reachability* on catalytic Turing machines in space $O(\lg n)$ with catalytic space $O(n^2 \lg n)$ and time $O(n^9)$.

- Problems like *directed st-reachability* [5], *distance* [56] which asks whether a given G (directed, undirected or even directed acyclic) contains a path of length at most k from s to t, are NL-complete i.e., no deterministic log-space algorithm is known. But in both the rotate and implicit models, we design log-space algorithms for them. Assuming $\mathsf{L} \neq \mathsf{NL}$, these results show that probably both our models with log-space are stronger than NL.
- The lex-DFS problem (both in undirected and directed graphs) is P-complete [54], and thus polylogarithmic space algorithms are unlikely to exist in the ROM model. But we show an $O(\lg n)$ space algorithm in the implicit model for lex-DFS. This implies that, probably the implicit model is even more powerful than the rotate model. It could even be possible that every problem in P can be computed using log-space in the implicit model. A result of somewhat similar flavor is obtained recently Buhrman et al. [18, 46] where they showed that any function in TC^1 can be computed using catalytic log-space, i.e., $\mathsf{TC}^1 \subseteq \mathsf{CSPACE}(\lg n)$. Note that TC^1 contains L, NL and even other classes that are conjectured to be different from L.
- For a large number of NP-hard graph problems, the best algorithms in ROM run in exponential time and polynomial space. We show that using just logarithmic amount of space, albeit using exponential time, we can design algorithms for those NP-hard problems in both of our models under some restrictions. This gives an exponential improvement over the ROM space bounds for these problems. In constrast, no NP-hard problem can be solved in the ROM model using $O(\lg n)$ bits unless P=NP. We discuss the details in the full version [21].

2 DFS algorithms in the rotate model

We first describe our space-efficient algorithms for DFS in the rotate model proving Theorem 1.

2.1 Proof of Theorem 1(a) for undirected graphs

We begin by describing our algorithm for undirected graphs, and later mention the changes required for directed graphs. In the normal exploration of DFS (see for example, Cormen et al. [29]) we use three colors. Every vertex v is white initially while it has not been discovered yet, becomes gray when DFS discovers v for the first time, and is colored black when it is finished i.e., all its neighbors have been explored completely.

We maintain a color array C of length n that stores the color of each vertex at any point in the algorithm. In the rest of the paper, when we say we scan the adjacency list of some vertex v, what we mean is, we create a temporary pointer pointing to the current first element of the list and move this temporary pointer until we find the desired element. Once we get that element we actually rotate the list so that the desired element now is at the front of the list. We start DFS at the starting vertex, say s, changing its color from white to gray in the color array C. Then we scan the adjacency list of s to find the first white neighbor, say w. We keep rotating the list to bring w to the front of s's adjacency list (as the one pointed to by the head $V[s]$), color w gray in the color array C and proceed to the next step (i.e. to explore w's adjacency list). This is the first *forward* step of the algorithm. In general, at any step during the execution of the algorithm, whenever we arrive at a gray vertex u

(either from one of u's black children or when u's color is changed from white to gray in the current step), we scan u's adjacency list to find the first white vertex. (i) If we find such a vertex, say v, then we rotate u's list to make v as the first element, and change the color of v to gray. (ii) If we do not find any white vertex, then we change the color of u to black, and *backtrack* to its parent. To identify u's parent, we use the following lemma.

▶ **Lemma 6.** (♠) *Suppose w is a node that just became* black. *Then its parent p is the unique vertex in w's list which is (a)* gray *and (b) whose current list has w in the first position.*

So, the parent can be found by scanning the w's list, to find a neighbor p that is colored gray such that the first element in p's list is u. This completes the description of the backtracking step. Once we backtrack to p, we find the next white vertex (as in the forward step) and continue until all the vertices of G are explored. Other than some constant number of variables, clearly the space usage is only for storing the color array C. Since C is of length n where each element has 3 possible values, C can be encoded using $n \lg 3 + O(\lg^2 n)$ bits, so that the i-th element in C can be read and updated in $O(1)$ time [32]. So overall space required is $n \lg 3 + O(\lg^2 n)$ bits. It is easy to see that at most two full rotations of each of the list may happen during the execution of the algorithm (first one to explore all the white neighbors and the second one to determine that there are no more white neighbors) resulting in a linear time lex-DFS algorithm. We discuss the lex-DFS algorithm for the directed graphs in the full version of the paper [21].

2.2 Proof of Theorem 1(b) for undirected graphs

To improve the space further, we replace the color array C with a bit array $visited[1, \ldots, n]$ which stores a 0 for an unvisited vertex (white), and a 1 for a visited vertex (gray or black). First we need a test similar to that in the statement of Lemma 6 without the distinction of gray and black vertices to find the parent of a node. Due to the invariant we have maintained, every internal vertex of the DFS tree will point to (i.e. have as first element in its list) its current last child. So the nodes that could potentially have node w in its first position are its parent, and any of its children which is a leaf. Hence we modify the forward step in the following way.

Whenever we visit an unvisited vertex v for the first time from another vertex u (hence, u is the parent of v in the DFS tree and u's list has v in the first position), we, as before, mark v as visited and in addition to that, we rotate v's list to bring u to the front (during this rotation, we do not mark any intermediate nodes as visited). Then we continue as before (by finding the first unvisited vertex and bringing it to the front) in the forward step. Now the following invariants are easy to see and are useful.

Invariants: During the exploration of DFS, in the (partial) DFS tree
1. any internal vertex has the first element in its list as its current last child; and
2. for any leaf vertex of the DFS tree, the first element in its list is its parent.

The first invariant is easy to see as we always keep the current explored vertex (child) as the first element in the list. For leaves, the first time we encounter them, we make its parent as the first element in the forward direction. Then we discover that it has no unvisited vertices in its list, and so we make a full rotation and bring the parent to the front again. The following lemma provides a test to find the parent of a node.

▶ **Lemma 7.** (♠) *Let w be a node that has just become* black. *Then its parent p is the* ***first*** *vertex x in w's list whose current adjacency list has w in the first position.*

Once we backtrack to p, we find the next white vertex, and continue until all the vertices of G are explored. Overall this procedure takes linear time. As we rotate the list to bring the parent of a node, before exploring its white neighbors, we are not guaranteed to explore the first white vertex in its original list, and hence we loose the lexicographic property. We provide our DFS algorithm for directed graphs in the full version [21].

2.3 Proof of Theorem 1(c) for undirected graphs

Now to decrease the space to $O(\lg n)$, we dispense with the color/visited array, and give tests to determine white, gray and black vertices. For now, assume that we can determine the color of a vertex. The forward step is almost the same as before except performing the update in the color array. I.e., whenever we visit a white vertex v for the first time from another vertex u (hence u is the parent of v), we rotate v's list to bring u to the front. Then we continue to find the first white vertex to explore. We maintain the following invariants. (i) any gray vertex has the first element in its list as its last child in the (partial) DFS tree; (ii) any black vertex has its parent as the first element in its list. We also store the depth of the current node in a variable d, which is incremented by 1 every time we discover a white vertex and decremented by 1 whenever we backtrack. We maintain the maximum depth the DFS has attained using a variable max. At a generic step during the execution of the algorithm, assume that we are at a vertex x's list, let p be x's parent and let y be a vertex in x's list. We need to determine the color of y and continue the DFS based on the color of y. We use the following characterization.

▶ **Lemma 8.** (♠) *Suppose the DFS has explored starting from a source vertex s, up to a vertex x at level d. Let p be x's parent. Note that both s and x are* gray *in the normal coloring procedure. Let max be the maximum level of any vertex in the partial DFS exploration. Let y be a vertex in x's list. Then,*

1. *y is* gray *(i.e., (x, y) is a back edge, and y is an ancestor of x) if and only if we can reach y from s by following through the* gray *child (which is at the front of a* gray *node's list) path in at most d steps.*
2. *y is* black *(i.e., (x, y) is a back edge, and x is an ancestor of y) if and only if*
   - *there is a path P of length at most $(max - d)$ from y to x (obtained by following through the first elements of the lists of every vertex in the path, starting from y), and*
   - *let z be the node before x in the path P. The node z appears after p in x's list.*
3. *y is* white *if y is not* gray *or* black.

Now, if we use the above claim to test for colors of vertices, testing for gray takes at most d steps. Testing for black takes at most $(max - d)$ steps to find the path, and at most $deg(x)$ steps to determine whether p appears before. Thus for each vertex in x's list, we spend time proportional to $max + deg(x)$. So, the overall runtime of the algorithm is $\sum_{v \in V} deg(v)(deg(v) + \ell) = O(m^2/n + m\ell)$, where ℓ is the maximum depth of DFS tree. Maintaining the invariants for the gray and black vertices are also straightforward. We provide the details of our log-space algorithm for directed graphs in the full version [21].

3 Simulation of algorithms for rotate model in the implicit model

The following result captures the overhead incurred while simulating any rotate model algorithm in the implicit model.

▶ **Theorem 9.** (♠) *Let D be the maximum degree of G. Then any algorithm running in $t(m,n)$ time in the* rotate *model can be simulated in the* implicit *model in (i) $O(D \cdot t(m,n))$ time when G is given in an adjacency list, and (ii) $O(\lg D \cdot t(m,n))$ time when G is given in an adjacency array. Furthermore, let $r_v(m,n)$ denote the number of rotations made in v's (whose degree is d_v) list, and $f(m,n)$ be the remaining number of operations. Then any algorithm running in $t(m,n) = f(m,n) + \sum_{v \in V} r_v(m,n)$ time in the* rotate *model can be simulated in the* implicit *model in (i) $O(f(m,n) + \sum_{v \in V} r_v(m,n) \cdot d_v)$ time when G is given in an adjacency list, and (ii) $O(f(m,n) + \sum_{v \in V} r_v(m,n) \lg d_v)$ time when G is given in an adjacency array.*

Most of our algorithms in the implicit model use these simulations often with some enhancements and tricks to obtain better running time bounds for some specific problems.

4 DFS algorithms in the implicit model – proof of Theorem 2

To obtain a lex-DFS algorithm, we implement the $O(\lg n)$-bit DFS algorithm in the rotate model, described in Section 2.3, with a simple modification. First, note that in this algorithm (in the rotate model), we bring the parent of a vertex to the front of its adjacency list (by performing rotations) when we visit a vertex for the first time. Subsequently, we explore the remaining neighbors of the vertex in the left-to-right order. Thus, for each vertex, if its parent in the DFS were at the beginning of its adjacency list, then this algorithm would result in a lex-DFS algorithm. Now, to implement this algorithm in the implicit model, whenever we need to bring the parent to the front, we simply bring it to the front without changing the order of the other neighbors. Subsequently, we simulate each rotation by moving all the elements in the adjacency list circularly. As mentioned in Section 3, this results in an algorithm whose running time is $O(\sum_{v \in V} d_v(d_v + \ell) \cdot d_v) = O(m^3/n^2 + \ell m^2/n)$ if the graph is given in an adjancecy list and in $O(\sum_{v \in V} d_v(d_v + \ell) \cdot \lg d_v) = O(m^2(\lg n)/n + m\ell \lg n))$ when the graph is given in the form of an adjacency array. This proves Theorem 2(a) for undirected graphs. The results for the directed case follow from simulating the corresponding results for the directed graphs.

To prove the result mentioned in Theorem 2(b), we implement the linear-time DFS algorithm of Theorem 1 for the rotate model that uses $n + O(\lg n)$ bits. This results in an algorithm that runs in $O(\sum_{v \in V} d_v^2 + n) = O(m^2/n)$ time (or in $O(\sum_{v \in V} d_v \lg d_v + n) = O(m \lg m + n)$ time, when the graph is given as an adjacency array representation), using $n + O(\lg n)$ bits. We reduce the space usage of the algorithm to $O(\lg n)$ bits by encoding the visited/unvisited bit for each vertex with degree at least 2 within its adjacency list (and not maintaining this bit for degree-1 vertices). We describe the details below.

Whenever a node is visited for the first time in the algorithm for the rotated list model, we bring its parent to the front of its adjacency list. In the remaining part of the algorithm, we process each of its other adjacent vertices while rotating the adjacency list, untill the parent comes to the front again. Thus, for each vertex v with degree d_v, we need to rotate v's adjacency list $O(d_v)$ times. In the implicit model, we also bring the parent to the front when a vertex is visited for the first time, for any vertex with degree at least 3. We use the second and third elements in the adjacency list to encode the visited/unvisited bit. But instead of rotating the adjacency list circularly, we simply scan through the adjacency list from left to right everytime we need to find the next unvisited vertex in its adjacency list. This requires $O(d_v)$ time for a vertex v with degree d_v. We show how to handle vertices with degree at most 2 separately.

As before, we can deal with the degree-1 vertices without encoding visited/unvisited bit as we encounter those vertices only once during the algorithm. For degree-2 vertices, we initially (at preprocessing stage) encode the bit 0 using the two elements in their adjacency arrays – to indicate that they are unvisited. When a degree-2 vertex is visited for the first time from a neighbor x, we move to its other neighbor – continuing the process as long as we encounter degree-2 vertices until we reach a vertex y with degree at least 3. If y is already visited, then we output the path consisting of all the degree-2 vertices and backtrack to x. If y is not visited yet, then we output the path up to y, and continue the search from y, and after marking y as visited. In both cases, we also mark all the degree-2 nodes as visited (by swapping the two elements in each of their adjacency arrays).

During the preprocessing, for each vertex with degree at least 3, we ensure that the second and third elements in its adjacency list encode the bit 0 (to mark it unvisited). We maintain the invariant that for any vertex with degree at least 3, as long as it is not visited, the second and third elements in its adjacency array encode the bit 0; and after the vertex is visited, its parent (in the DFS tree) is at the front of its adjacency array, and the second and third elements in its adjacency array encode the bit 1. Thus, when we visit a node v with degree at least 3 for the first time, we bring its parent to the front, and then swap the second and third elements in the adjacency list, if needed, to mark it as visited. The total running time of this algorithm is bounded by $\sum_{v \in V} d_v^2 = O(m^2/n)$.

We can implement the above DFS algorithm even faster when the input graph is given in an adjacency array representation. We deal with vertices with degree at most 2 exactly as before. For a vertex v with degree at least 3, we bring its parent to the front and swap the second and third elements to mark the node as visited (as before) whenever v is visited for the first time. We then sort the remaining elements, if any, in the adjacency array, in-place (using the linear-time in-place radix sort algorithm [42]), and implement the rotations on the remaining part of the array as described in Section 3. The total running time of this algorithm is bounded by $\sum_{v \in V} d_v \lg d_v = O(m \lg m + n)$. This completes the proof of Theorem 2(b).

5 Concluding remarks

Our initial motivation was to get around the limitations of ROM to obtain a reasonable model for graphs in which we can obtain space efficient algorithms. We achieved that by introducing two new frameworks and obtained efficient (of the order of $O(n^3 \lg n)$) algorithms using $O(\lg n)$ bits of space for fundamental graph search procedures. We also discussed various applications of our DFS/BFS results, and it is not surprising that many corollaries would follow as they are the backbone of many algorithms. We showed that some of these results also translate to improved space efficient algorithms in ROM (by simulating the rotate model algorithms in ROM with a pointer per list). With some effort, we can obtain log space algorithm for MST. These results can be contrasted with the state of the art results in ROM that take almost linear bits for some of these problems other than having large runtime bounds. We believe that our work is the first step towards designing efficient in-place graph algorithms and it will inspire further investigation into designing such algorithms for other graph problems. One future direction would also be to improve the running times of our algorithms. As in the case of most of the earlier space-efficient graph algorithms, we only consider adjacency list and array representation. It's not clear how to define in-place model for adjacency matrix. Another challenging direction would be to design efficient algorithms that also restore the initial input representation at the end of the execution of the algorithm.

Surprisingly we could design log-space algorithm for some P-complete problems, and so it is important to understand the power of our models. Towards that we discovered that we can even obtain log-space algorithms for some NP-hard graph problems. More specifically, we defined *graph subset problems* and obtained log-space exponential time algorithms for problems belonging to this class in [21]. One interesting future direction would be to determine the exact computational power of these models along with exploring the horizon of interesting complexity theoretic consequences of problems in these models. Unlike the ROM model, it's not clear how one can define an in-place model which is closed under composition. We leave this as a challenging open problem.

References

1. A. Aggarwal and R. J. Anderson. A random NC algorithm for depth first search. *Combinatorica*, 8(1):1–12, 1988. doi:10.1007/BF02122548.
2. A. Aggarwal, R. J. Anderson, and M. Kao. Parallel depth-first search in general directed graphs. *SIAM J. Comput.*, 19(2):397–409, 1990. doi:10.1137/0219025.
3. N. Alon, Y. Matias, and M. Szegedy. The space complexity of approximating the frequency moments. *J. Comput. Syst. Sci.*, 58(1):137–147, 1999. doi:10.1006/jcss.1997.1545.
4. R. J. Anderson and E. W. Mayr. Parallelism and the maximal path problem. *Inf. Process. Lett.*, 24(2):121–126, 1987. doi:10.1016/0020-0190(87)90105-0.
5. S. Arora and B. Barak. *Computational Complexity - A Modern Approach*. Cambridge University Press, 2009. URL: http://www.cambridge.org/catalogue/catalogue.asp?isbn=9780521424264.
6. T. Asano, K. Buchin, M. Buchin, M.Korman, W. Mulzer, G. Rote, and A. Schulz. Reprint of: Memory-constrained algorithms for simple polygons. *Comput. Geom.*, 47(3):469–479, 2014. doi:10.1016/j.comgeo.2013.11.004.
7. T. Asano, T. Izumi, M. Kiyomi, M. Konagaya, H. Ono, Y. Otachi, P. Schweitzer, J. Tarui, and R. Uehara. Depth-first search using O(n) bits. In *25th ISAAC*, pages 553–564, 2014.
8. T. Asano, D. G. Kirkpatrick, K. Nakagawa, and O. Watanabe. $\tilde{O}(\sqrt{n})$-space and polynomial-time algorithm for planar directed graph reachability. In *39th MFCS LNCS 8634*, pages 45–56, 2014. doi:10.1007/978-3-662-44465-8_5.
9. T. Asano, W. Mulzer, G. Rote, and Y. Wang. Constant-work-space algorithms for geometric problems. *JoCG*, 2(1):46–68, 2011. URL: http://jocg.org/index.php/jocg/article/view/30.
10. N. Banerjee, S. Chakraborty, and V. Raman. Improved space efficient algorithms for BFS, DFS and applications. In *22nd COCOON*, 2016. URL: http://arxiv.org/abs/1606.04718.
11. N. Banerjee, S. Chakraborty, V. Raman, S. Roy, and S. Saurabh. Time-space tradeoffs for dynamic programming in trees and bounded treewidth graphs. In *21st COCOON*, volume 9198, pages 349–360. springer, LNCS, 2015.
12. L. Barba, M. Korman, S. Langerman, K. Sadakane, and R. I. Silveira. Space-time trade-offs for stack-based algorithms. *Algorithmica*, 72(4):1097–1129, 2015. doi:10.1007/s00453-014-9893-5.
13. G. Barnes, J. Buss, W. Ruzzo, and B. Schieber. A sublinear space, polynomial time algorithm for directed s-t connectivity. *SIAM J. Comput.*, 27(5):1273–1282, 1998. doi:10.1137/S0097539793283151.
14. Paul Beame. A general sequential time-space tradeoff for finding unique elements. *SIAM J. Comput.*, 20(2):270–277, 1991. doi:10.1137/0220017.
15. A. Borodin and S. A. Cook. A time-space tradeoff for sorting on a general sequential model of computation. *SIAM J. Comput.*, 11(2):287–297, 1982. doi:10.1137/0211022.

16 A. Borodin, M. J. Fischer, D. G. Kirkpatrick, N. A. Lynch, and M. Tompa. A time-space tradeoff for sorting on non-oblivious machines. *J. Comput. Syst. Sci.*, 22(3):351–364, 1981. doi:10.1016/0022-0000(81)90037-4.

17 H. Brönnimann, T. M. Chan, and E. Y. Chen. Towards in-place geometric algorithms and data structures. In *Proceedings of the 20th ACM Symposium on Computational Geometry, Brooklyn, New York, USA, June 8-11, 2004*, pages 239–246, 2004. doi:10.1145/997817.997854.

18 H. Buhrman, R. Cleve, M. Koucký, B. Loff, and F. Speelman. Computing with a full memory: catalytic space. In *Symposium on Theory of Computing, STOC 2014, New York, NY, USA, May 31 - June 03, 2014*, pages 857–866, 2014. doi:10.1145/2591796.2591874.

19 H. Buhrman, M.l Koucký, B. Loff, and F. Speelman. Catalytic space: Non-determinism and hierarchy. In *33rd STACS 2016, February 17-20, 2016, Orléans, France*, pages 24:1–24:13, 2016. doi:10.4230/LIPIcs.STACS.2016.24.

20 D. Chakraborty, A. Pavan, R. Tewari, N. V. Vinodchandran, and L. Yang. New time-space upperbounds for directed reachability in high-genus and h-minor-free graphs. In *FSTTCS*, pages 585–595, 2014. doi:10.4230/LIPIcs.FSTTCS.2014.585.

21 S. Chakraborty, A. Mukherjee, V. Raman, and S. R. Satti. Frameworks for designing in-place graph algorithms. *CoRR*, abs/1711.09859, 2017. arXiv:1711.09859.

22 S. Chakraborty, V. Raman, and S. R. Satti. Biconnectivity, chain decomposition and st-numbering using O(n) bits. In *27th ISAAC*, pages 22:1–22:13, 2016. doi:10.4230/LIPIcs.ISAAC.2016.22.

23 S. Chakraborty, V. Raman, and S. R. Satti. Biconnectivity, st-numbering and other applications of DFS using O(n) bits. *J. Comput. Syst. Sci.*, 90:63–79, 2017. doi:10.1016/j.jcss.2017.06.006.

24 S. Chakraborty and S. R. Satti. Space-efficient algorithms for maximum cardinality search, stack bfs, queue BFS and applications. In *Computing and Combinatorics - 23rd International Conference, COCOON 2017, Hong Kong, China, August 3-5, 2017, Proceedings*, pages 87–98, 2017. doi:10.1007/978-3-319-62389-4_8.

25 T. M. Chan and E. Y. Chen. Multi-pass geometric algorithms. *Discrete & Computational Geometry*, 37(1):79–102, 2007. doi:10.1007/s00454-006-1275-6.

26 T. M. Chan, J. I. Munro, and V. Raman. Faster, space-efficient selection algorithms in read-only memory for integers. In *Algorithms and Computation - 24th International Symposium, ISAAC 2013, Hong Kong, China, December 16-18, 2013, Proceedings*, pages 405–412, 2013. doi:10.1007/978-3-642-45030-3_38.

27 T. M. Chan, J. I. Munro, and V. Raman. Selection and sorting in the "restore" model. In *25th-SODA*, pages 995–1004, 2014. doi:10.1137/1.9781611973402.74.

28 S. A. Cook and C. Rackoff. Space lower bounds for maze threadability on restricted machines. *SIAM J. Comput.*, 9(3):636–652, 1980. doi:10.1137/0209048.

29 T. H. Cormen, C. E. Leiserson, R. L. Rivest, and C. Stein. *Introduction to Algorithms (3. ed.)*. MIT Press, 2009. URL: http://mitpress.mit.edu/books/introduction-algorithms.

30 O. Darwish and A. Elmasry. Optimal time-space tradeoff for the 2d convex-hull problem. In *22th ESA*, pages 284–295, 2014. doi:10.1007/978-3-662-44777-2_24.

31 S. Datta, N. Limaye, P. Nimbhorkar, T. Thierauf, and F. Wagner. Planar graph isomorphism is in log-space. In *24th CCC*, pages 203–214, 2009. doi:10.1109/CCC.2009.16.

32 Y. Dodis, M. Patrascu, and M. Thorup. Changing base without losing space. In *Proceedings of the 42nd ACM Symposium on Theory of Computing (STOC)*, pages 593–602, 2010. doi:10.1145/1806689.1806770.

33 J. Edmonds, C. K. Poon, and D. Achlioptas. Tight lower bounds for st-connectivity on the NNJAG model. *SIAM J. Comput.*, 28(6):2257–2284, 1999. doi:10.1137/S0097539795295948.

34 M. Elberfeld, A. Jakoby, and T. Tantau. Logspace versions of the theorems of bodlaender and courcelle. In *51th FOCS*, pages 143–152, 2010. doi:10.1109/FOCS.2010.21.

35 M. Elberfeld and K. Kawarabayashi. Embedding and canonizing graphs of bounded genus in logspace. In *Symposium on Theory of Computing, STOC 2014, New York, NY, USA, May 31 - June 03, 2014*, pages 383–392, 2014. doi:10.1145/2591796.2591865.

36 M. Elberfeld and P. Schweitzer. Canonizing graphs of bounded tree width in logspace. In *33rd Symposium on Theoretical Aspects of Computer Science, STACS 2016, February 17-20, 2016, Orléans, France*, pages 32:1–32:14, 2016. doi:10.4230/LIPIcs.STACS.2016.32.

37 A. Elmasry, T. Hagerup, and F. Kammer. Space-efficient basic graph algorithms. In *32nd STACS*, pages 288–301, 2015. doi:10.4230/LIPIcs.STACS.2015.288.

38 A. Elmasry, D. D. Juhl, J. Katajainen, and S. R. Satti. Selection from read-only memory with limited workspace. *Theor. Comput. Sci.*, 554:64–73, 2014. doi:10.1016/j.tcs.2014.06.012.

39 J. Feigenbaum, S. Kannan, A. McGregor, S. Suri, and J. Zhang. On graph problems in a semi-streaming model. *Theor. Comput. Sci.*, 348(2-3):207–216, 2005. doi:10.1016/j.tcs.2005.09.013.

40 G. Franceschini and J. Ian Munro. Implicit dictionaries with $O(1)$ modifications per update and fast search. In *Proceedings of the Seventeenth Annual ACM-SIAM Symposium on Discrete Algorithms (SODA)*, pages 404–413, 2006.

41 G. Franceschini and S. Muthukrishnan. In-place suffix sorting. In *Automata, Languages and Programming, 34th International Colloquium, ICALP 2007, Wroclaw, Poland, July 9-13, 2007, Proceedings*, pages 533–545, 2007. doi:10.1007/978-3-540-73420-8_47.

42 G. Franceschini, S. Muthukrishnan, and M. Patrascu. Radix sorting with no extra space. In *Algorithms - ESA 2007, 15th Annual European Symposium, Eilat, Israel, October 8-10, 2007, Proceedings*, pages 194–205, 2007. doi:10.1007/978-3-540-75520-3_19.

43 G. N. Frederickson. Upper bounds for time-space trade-offs in sorting and selection. *J. Comput. Syst. Sci.*, 34(1):19–26, 1987. doi:10.1016/0022-0000(87)90002-X.

44 T. Hagerup and F. Kammer. Succinct choice dictionaries. *CoRR*, abs/1604.06058, 2016. URL: http://arxiv.org/abs/1604.06058, arXiv:1604.06058.

45 F. Kammer, D. Kratsch, and M. Laudahn. Space-efficient biconnected components and recognition of outerplanar graphs. In *41st MFCS*, 2016.

46 M. Koucký. Catalytic computation. *Bulletin of the EATCS*, 118, 2016. URL: http://eatcs.org/beatcs/index.php/beatcs/article/view/400.

47 T. W. Lai and D. Wood. Implicit selection. In *SWAT 88, 1st Scandinavian Workshop on Algorithm Theory, Halmstad, Sweden, July 5-8, 1988, Proceedings*, pages 14–23, 1988. doi:10.1007/3-540-19487-8_2.

48 A. McGregor. Graph stream algorithms: a survey. *SIGMOD Record*, 43(1):9–20, 2014. doi:10.1145/2627692.2627694.

49 J. I. Munro and M. Paterson. Selection and sorting with limited storage. *Theor. Comput. Sci.*, 12:315–323, 1980. doi:10.1016/0304-3975(80)90061-4.

50 J. I. Munro and V. Raman. Selection from read-only memory and sorting with minimum data movement. *Theor. Comput. Sci.*, 165(2):311–323, 1996. doi:10.1016/0304-3975(95)00225-1.

51 J. Ian Munro. An implicit data structure supporting insertion, deletion, and search in $O(\log^2 n)$ time. *J. Comput. Syst. Sci.*, 33(1):66–74, 1986. doi:10.1016/0022-0000(86)90043-7.

52 J. Pagter and T. Rauhe. Optimal time-space trade-offs for sorting. In *39th Annual Symposium on Foundations of Computer Science, FOCS '98, November 8-11, 1998, Palo Alto, California, USA*, pages 264–268, 1998. doi:10.1109/SFCS.1998.743455.

53 J. H. Reif. Symmetric complementation. *J. ACM*, 31(2):401–421, 1984. doi:10.1145/62.322436.

54 J. H. Reif. Depth-first search is inherently sequential. *Inf. Process. Lett.*, 20(5):229–234, 1985. doi:10.1016/0020-0190(85)90024-9.

55 O. Reingold. Undirected connectivity in log-space. *J. ACM*, 55(4), 2008. doi:10.1145/1391289.1391291.

56 T. Tantau. Logspace optimization problems and their approximability properties. *Theory Comput. Syst.*, 41(2):327–350, 2007. doi:10.1007/s00224-007-2011-1.

57 M. Tompa. Two familiar transitive closure algorithms which admit no polynomial time, sublinear space implementations. *SIAM J. Comput.*, 11(1):130–137, 1982. doi:10.1137/0211010.