Optimal In-place Algorithms for Basic Graph Problems

Sankardeep Chakraborty[1], Kunihiko Sadakane[2(✉)], and Srinivasa Rao Satti[3]

[1] National Institute of Informatics, Tokyo, Japan
sankar@nii.ac.jp
[2] The University of Tokyo, Tokyo, Japan
sada@mist.i.u-tokyo.ac.jp
[3] Seoul National University, Seoul, South Korea
ssrao@cse.snu.ac.kr

Abstract. We present linear time *in-place* algorithms for several fundamental graph problems including the well-known graph search methods (like depth-first search, breadth-first search, maximum cardinality search), connectivity problems (like biconnectivity, 2-edge connectivity), decomposition problem (like chain decomposition) among various others, improving the running time (by polynomial multiplicative factor) of the recent results of Chakraborty et al. [ESA, 2018] who designed $O(n^3 \lg n)$ time in-place algorithms for some of the above mentioned problems. The running times of all our algorithms are essentially optimal as they run in linear time. One of the main ideas behind obtaining these algorithms is the detection and careful exploitation of sortedness present in the input representation for any graph without loss of generality. This observation alone is powerful enough to design some basic linear time in-place algorithms, but more non-trivial graph problems require extra techniques which, we believe, may find other applications while designing in-place algorithms for different graph problems in future.

1 Introduction

Inspired by the rapid growth of humongous data set ("big data phenomenon"), *space efficient algorithms* are becoming increasingly more crucial than ever before. The dire need of such algorithms is also propelled by the pervasive usage of small specialized handheld devices and embedded systems which come equipped with tiny memory. To design such algorithms, a vast array of computational models have already been proposed in the literature. In what follows, we briefly mention a few of them in the order they are historically developed.

In the *read-only memory* model (henceforth ROM) where the input is read-only, output is write only, and a limited sized random access read/write work space is available, researchers have designed space efficient algorithms

The full version of this paper appears as [15]. The work of the first author is supported by JSPS KAKENHI Grants Number 18H05291.

L. Gąsieniec et al. (Eds.): IWOCA 2020, LNCS 12126, pp. 126–139, 2020.
https://doi.org/10.1007/978-3-030-48966-3_10

for selection and sorting [18,26,33,39,40], problems in computational geometry [2,4,6,17,22], and graphs [3,5,10,12–14,25] among various others. In the *in-place* model, it is assumed that the input elements are given in an array, and the algorithm may use the input array as working space, hence the algorithm is allowed to modify the array during its execution. However, at any point during 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 entire execution of the algorithm is limited to $O(\lg n)$ bits only. A prime example of an in-place algorithm is the classic heap-sort. Other than in-place sorting [32], searching [30,37] and selection [36], many in-place algorithms were designed in areas such as computational geometry [8] and string algorithms [31]. A very recent addition to this long list is the in-place algorithms for the graph problems [11]. Other than these, researchers have also designed space efficient algorithms in *(semi)-streaming* models [1,29,39] and recently introduced *restore* [19] and *catalytic-space* [9] models.

Previous Work on Space Efficient Graph Algorithms. Inspired by the pervasive practical applications of the fundamental graph algorithms, recently there has been a surge of interest in improving the space complexity of graph algorithms without paying too much penalty in the running time. Thus the goal is to design space-efficient yet reasonably time-efficient graph algorithms on the ROM. Generally most of the standard implementations of classical graph algorithms take linear or near-linear running time and use $O(n \lg n)$ (or sometimes $O(m \lg n)$ for graphs with n vertices and m edges) bits. A recent series of papers [3,5,13,16,25] with this point of view showed such results for a vast array of basic graph problems, namely depth-first search (henceforth DFS), breadth-first search (henceforth BFS), minimum spanning tree (henceforth MST), (strong) connectivity, topological sorting, recognizing chordal graphs, biconnectivity, *st*-numbering, shortest path and many others.

Even though these results are still both time and space efficient, they still require $\Theta(n)$ bits for most of important graph algorithms, and this is a major concern in places with severe space constraints. In order to break this inherent space bound barrier and still obtain reasonable time efficiency, Chakraborty et al. [11] initiated a systematic study of designing efficient *in-place* (i.e., using $O(\lg n)$ bits of extra space other than the input space) algorithms for graph problems by defining a new framework which is a slight relaxation of the ROM. Using this framework they designed in-place DFS, BFS, MST, reachability algorithms taking time $O(n^3 \lg n)$. Despite being optimal in space usage, observe that these results still leave a polynomial gap in the running time from the optimal value. In this work, we essentially obtain the best of the both worlds by closing this gap. More specifically, we show how one can design optimal in-place algorithms i.e., $O(m+n)$ time and using $O(\lg n)$ bits of extra space, for several of these (and a lot more) basic graph algorithms in this work. Recently Kammer et al. [34] also considered a similar model where they showed efficient in-place algorithms for DFS, unordered-BFS (will be defined shortly) only.

In-place Model for Graph Algorithms and Input Representations. Before explaining our in-place algorithms and stating main results, in this section we first describe the input graph representation. Note that, as in the case of the standard in-place model, we need to ensure that the graph (adjacency) structure must remain intact throughout the entire execution of the algorithm. Let $G = (V, E)$ be the input graph with $n = |V|$, $m = |E|$, and as usual let $V = \{1, 2, \cdots, n\}$ denote the vertex set of G. We assume that the input graph is given in the standard adjacency array format, and throughout this paper, we refer to this array as Z. More specifically, it is an array having size $(n + m + 1)$ ($(n+2m+1)$ resp.) words for directed (undirected resp.) graphs where $Z[1]$ stores the number of vertices in G, the next n entries (which we refer to as the *offsets* part of Z) store n pointers (one per vertex) pointing to the location in Z of the last neighbor for each vertex, and finally the last m ($2m$ for undirected graphs) entries are reserved for the edges of G. At this point, we should emphasize a small, yet important, technical detail. The Z array can be thought of as a single bit array as follows. For a directed graph G, the array Z is a concatenation of $Z[1]$ of length $\lceil \lg n \rceil$ bits, $Z[2] \dots Z[n+1]$ of length $\lceil \lg m \rceil$ bits each[1], and finally $Z[n+2] \dots Z[n+m+1]$ of length $\lceil \lg n \rceil$ bits each. For undirected graphs, only the second part changes to size $\lceil \lg m \rceil + 1$ bits (instead of $\lceil \lg m \rceil$) each. Thus, if we just remember the boundaries, we know exactly how many bits we need to read in order to extract useful information from the relevant parts of Z. For the sake of simplicity, we drop the ceiling notations from now on. Moreover, throughout this paper, it should be clear from the context the word size depending on which part of Z we are currently working on. See Fig. 1 for an example. Note that this representation implicitly captures the degree information for every vertex in G. Given this format, we say an algorithm $\mathcal{A}$ is an *in-place* algorithm if $\mathcal{A}$ (a) may modify any part of Z during its execution, (b) retains all the initial elements of Z (in any order) when it finishes execution; and (c) uses just $O(\lg n)$ bits of extra space. Our goal is to design such algorithms in this paper for a vast array of fundamental graph problems.

In this paper we assume the standard word RAM model of computation. 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.

Graph Terminology and Notations. In general we will assume the knowledge of basic graph theoretic terminology as given in [23] and basic graph algorithms as given in [21]. Still here we collect all the necessary graph theoretic definitions that will be used throughout the paper for quick reference and making the paper self-contained. For BFS traversal that we study here, there are two versions studied in the literature. In the *ordered* BFS (sometimes also known as queue BFS [16]), vertices are extracted from the queue in the first in first out (FIFO) order, whereas in the *unordered* BFS [5], vertices can be taken out from the

[1] Note that it is enough to store the offset values starting from 0, since we can add $n + 1$ to the offset value to find the corresponding location in Z; hence the offset values can be stored using $\lceil \lg m \rceil$ bits.

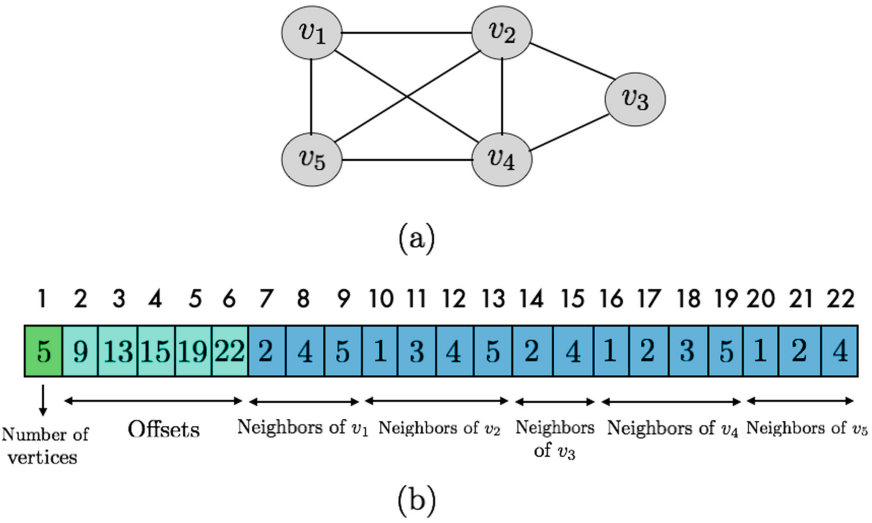

Fig. 1. (a) An undirected graph G with 5 vertices and 8 edges. (b) The standard adjacency array representation of G. To avoid cluttering the diagram, we drop the superscript v from the vertex labels while referring to them as neighbors.

queue in any order as long as no elements are extracted from a higher level of the BFS tree before finishing all the vertices from a lower level of the tree. In this paper, by a BFS/DFS traversal of the input graph G, as in [3,5,13,16] 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. Tarjan et al. [45] defined another method called maximum cardinality search (MCS) and used this to give a recognition algorithms for chordal graphs. MCS works as follows: assuming that every vertex is unnumbered at the beginning, at each iteration of the execution of MCS, an unnumbered vertex that is adjacent to the largest number of numbered vertices is chosen (breaking the ties arbitrarily), and is numbered with the next available label. Thus, the output of the MCS algorithm is a numbering of the vertices from 1 to n.

A cut vertex in an undirected graph G is a vertex v that when removed (along with its incident edges) from a graph creates more components than previously in the graph. A (connected) graph with at least three vertices is biconnected if and only if it has no cut vertex. Similarly in an undirected graph G, a bridge is an edge that when removed, creates more components than previously in the graph. A connected graph with at least two vertices is 2-edge-connected if and only if it has no bridge. Given a biconnected graph G, and two distinguished vertices s and t in V such that $s \neq t$, st-numbering is a numbering of the vertices of the graph so that s gets the smallest number, t gets the largest and every other vertex is adjacent both to a lower-numbered and to a higher-numbered vertex i.e., a numbering $s = v_1, v_2, \cdots, v_n = t$ of the vertices of G is called an st-numbering, if for all vertices $v_j, 1 < j < n$, there exist $1 \leq i < j < k \leq n$

such that $\{v_i, v_j\}, \{v_j, v_k\} \in E$. It is well-known that G is biconnected if and only if, for every edge $\{s, t\} \in E$, it has an st-numbering. A topological sort of a directed acyclic graph (DAG) gives a linear ordering of its vertices such that for every directed edge $(u, v) \in E$ from vertex u to vertex v, u comes before v in the ordering. A minimum spanning tree (MST) is a subset of the edges of a connected, edge-weighted undirected graph that connects all the vertices together, without any cycles and with the minimum possible total edge weight. That is, it is a spanning tree whose sum of edge weights is as small as possible.

Our Main Results and Organization of the Paper. In Sect. 2.1 we start by designing a linear time in-place procedure to obtain linear bits of additional free space inside the offsets part of the adjacency array. Using this, we can already show an improved set of algorithms for (a strict superset of) problems that Chakraborty et al. [16] considered (for example, DFS, unordered BFS and MST), but this algorithms are still not optimal as they are at least polylog multiplicative factor away from linear running time. Towards obtaining optimal linear time in-place algorithms, we first provide an improved linear time in-place routine to obtain almost $n \lg n$ additional free bits of space inside the offsets part, which is what we use crucially along with other additional ideas to show the following main result of this paper in Sect. 2.2.

Theorem 1. *Using linear time in the in-place model, one can*

1. *traverse the vertices of any graph in (un)ordered* BFS *and* DFS *manner,*
2. *recognize bipartite graphs, and compute connected components,*
3. *report the vertices of a* DAG *in topologically sorted order,*
4. *obtain a maximum cardinality search ordering of any graph,*
5. *output an st-numbering of given biconnected graph, given two vertices s and t,*
6. *perform a chain decomposition of any undirected graph, and*
7. *determine whether any given undirected graph G is biconnected (and/or* 2*-edge connected resp.) and if not, we can also compute and report all the cut vertices (bridges resp.) of G.*

*Also, given an undirected edge-weighted (where weights are bounded by some polynomial in n) graph G, we can find a minimum spanning tree (*MST*) of G in $O(m \lg n)$ time in-place.*

Techniques. All the results of our paper stem from the following very simple yet absolutely crucial observation: *numbers in sorted order have less entropy than in any arbitrary order*. More specifically, assuming we have n numbers from a universe of size m, when these numbers are in any arbitrary order their binary entropy is $n \lg m$ but when they are in sorted order, binary entropy becomes $n \lg m - \Theta(n \lg n)$. This clearly indicates that we can exploit the sorted structure assumption to gain some additional space. Now, note that, without loss of any generality, by construction, the *offsets* part of the adjacency array Z for any given graph G is sorted. Thus, we can use the above mentioned idea in the offsets part of Z to gain some free space which is what we use finally to design our optimal in-place graph algorithms. Towards this, we also have to handle several other key technical issues which we describe in respective sections in detail.

2 Exploiting Input Redundancy to Create Working Space

In this section, we describe how one can exploit the redundancy in the input representation to save almost $n \lg n$ bits, which can then be used as part of the working space for a graph algorithm.

2.1 Saving Linear Bits and Its Applications

As a warm up, we start by showing how we can squeeze in linear sized free bits inside the offsets part of Z while still being able to access any element inside the offsets part in $O(1)$ time, as well as returning to the original configuration of the offsets part of Z before freeing linear bits. Towards this, we first reprove the following lemma, which is essentially same as [34, Lemma 5]. See the full version [15] for a proof.

Lemma 1. *Given a sorted list of n integers from the universe $[0, m-1]$, it can be represented either simply as an array $A[1...n]$ with the integers in sorted order or as an array of n integers, such that for some fixed constant $c > 1$, the last cn bits of this array are all zero. Moreover, there exists an in-place $O(n)$ time algorithm for switching between both these formats.*

The above lemma alone is powerful enough to help us design in-place algorithms (albeit with sub-optimal time complexity as we will see shortly) for a variety of fundamental graph algorithms. In the full version [15] of this paper, we describe how to obtain efficient in-place algorithms for a variety of graph algorithms, using the above lemma. The main idea is to simulate the corresponding ROM algorithms in the in-place model. Next, we further improve the running times to optimal, by providing an improved version of Lemma 1.

2.2 Saving $n \lg n - 2n$ Bits

In what follows, we show how one can improve Lemma 1 so that almost $n \lg n$ bits become free to be used, and using this we will design optimal in-place algorithms for the above mentioned graph problems. Our main result can be described as follows:

Theorem 2. *Given a sorted list of n integers from the universe $[0, m-1]$, it can be represented either simply as an array $A[1...n]$ with the integers in sorted order or as an array of n integers, such that the last $n \lg n - 2n$ bits of this array are all zero. Moreover, there exists an in-place $O(n)$ time algorithm for switching between both these formats.*

Proof. One can easily obtain the space bound mentioned in the second representation by applying the Elias-Fano encoding [24,28] on the array A. But to implement this encoding in-place, we apply this encoding in two steps.

We first split the array A into two subarrays of size $n/2$ each (assume, for simplicity, that n is even) - call them A_1 and A_2. One can replace the most significant $\lg n$ bits of each of the elements in A_1 by a bit vector, say B, length

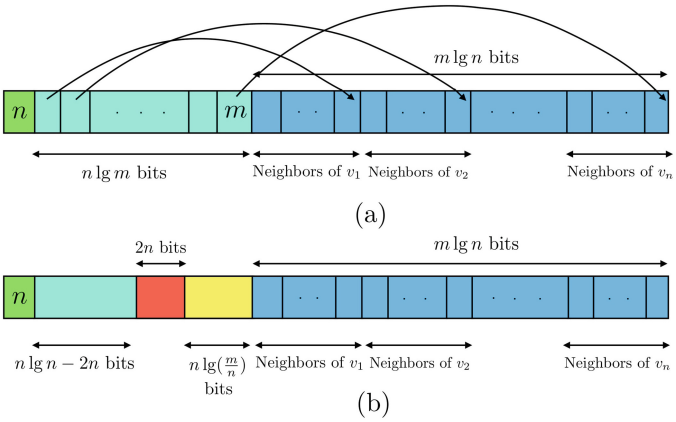

Fig. 2. (a) General adjacency array structure Z of a given input directed graph. (b) Configuration of Z after freeing $n \lg n - 2n$ bits in the offsets part of Z.

$n + n/2$, using the Elias-Fano encoding. To store B (of length $3n/2$), we first replace the most significant 3 bits of each of the elements in A_2 by storing 8 positions into the array A_2 (using Lemma 1, with $c = 3$). We store the bit vector B inside the most-significant 3 bits of every element of A_2, and compact the remaining (least-significant $\lg m - \lg n$) bits of every element in A_1 into a consecutive chunk of $(n/2)\lg(m/n)$ bits in A_1, so that the first $(n/2)\lg n$ bits of A_1 is free (i.e., filled with all zeros). We now copy the bit vector B into this free space, and restore the 3 most significant bits of all the elements of A_2. We now replace the most-significant $\lg n$ bits of each element in A_2 by a bit vector C of length $3n/2$, and store it inside free space in A_1 (here, we assume that $3n \leq (n/2)\lg n$), and compact the remaining (least-significant $\lg m - \lg n$) bits into a consecutive chunk of $(n/2)\lg(m/n)$ bits in A_2. Finally, we copy all the lower order bits (of total length $n\lg(m/n)$ bits) into a single chunk, and also merge the two bit vectors of length $3n/2$ each into a single bit vector of length $2n$. Thus the array A is replaced by a total of $n\lg(m/n) + 2n$ bits, giving a free space of $n\lg n - 2n$ bits. These steps can be essentially performed in reverse order to restore the original representation from the second representation. To support the operation of accessing the i-th element of A in $O(1)$ time, we can store an additional $o(n)$-bit auxiliary structure that support the *rank* and *select* operations [20,38] on the $2n$ bit sequence, which can then be used to access the most-significant $\lg n$ bits of any element in $O(1)$ time. The remaining $\lg m - \lg n$ bits can be simply read from the array of values stored in the second representation. See Fig. 2 for a visual description of the final outcome of application of this theorem.

3 Optimal In-place Graph Algorithms

In this section, we show how one can use Theorem 2 for solving the graph problems mentioned before. Before giving specific details, we would like to sketch the

general pattern for designing optimal in-place algorithms for some of these graph problems. Given the adjacency array representation (as in Z) of the input graph G, we now first apply Theorem 2 on the offsets part of Z to make $n \lg n - 2n$ bits free. Now the classical linear time algorithms [21,27,42–45] for these problems typically take $cn \lg n + dn$ bits where both the constants c and d are at most 2. Hence, our idea is to run these algorithms as it is but in some constant number of phases. More specifically, we store only, say $n/3$ vertices, explicitly at any point of time during the execution of these algorithms, and when these vertices are taken care of by the respective algorithms, we refresh the data structures by initiating it with a new set of $n/3$ vertices and proceed again till we exhaust all the vertices, thus, the entire algorithm would finish in three phases ultimately. Now the exact details of refreshing the data structure with a new set of vertices and start the algorithm again where it left off depends on specific problems. This idea would work for most of the algorithms that we discuss in this paper except a few important ones. More specifically, a few of the algorithms for those graph problems are two (or more) pass algorithms, i.e., in the first pass it computes some function which is what used in the second pass to solve the problem finally, for example, chain decomposition, biconnectivity etc. For these kinds of algorithms, it seems hard to make them work using the previously described constant phase algorithmic idea. Thus, we handle them differently by first proving some related lemmata which might be of independent interest, and then use these lemmata to design in-place algorithms for these graph problems. We discuss these after giving proofs for the algorithms which we can handle in constant phases only. In what follows we provide the proofs of linear time in-place algorithms for DFS and its applications, especially chain decomposition, biconnectivity, 2-edge connectivity, and also develop/prove the necessary ideas for these algorithms. The missing proofs of Theorem 1 cen be found in the full version [15].

The classical implementation of DFS (see for example, Cormen et al. [21]) uses three colors and a stack to traverse the whole graph. More specifically, every vertex v is white initially while it has not been discovered yet, becomes grey when DFS discovers v for the first time and pushes on the stack, and is colored black when it is finished i.e., all its neighbors have been explored completely, and it leaves the stack. The algorithm maintains a color array C of length $O(n)$ bits that stores the color of each vertex at any point in the algorithm, along with a stack (which could grow to $O(n \lg n)$ bits) for storing all the grey vertices at any point during the execution. Our idea is to run essentially the same DFS algorithm but we limit the stack size so that it contains at most $n/2$ latest grey vertices all the time. More specifically, whenever the stack grows to have more than $n/2$ vertices, we delete the bottom most vertex from the stack so that above invariant is always maintained along with storing the last such vertex to be deleted in order to enforce the invariant. At some point during the execution of the algorithm, when we arrive at a vertex v such that none of v's neighbors are white, then we color the vertex v as black, and we pop it from the stack. If the stack is still non-empty, then the parent of v (in the DFS tree) would be at the top of the stack, and we continue the DFS from this vertex. On the other hand,

if the stack becomes empty after removing v, we need to reconstruct it to the state such that it holds the last $n/2$ grey vertices after all the pops done so far. We refer to this phase of the algorithm as reconstruction step. For this, using ideas from [3,25], we basically repeat the same algorithm but with one twist which also enables us now to skip some of the vertices during this reconstruction phase. In detail, we again start with an empty stack, insert the root s first and scan its adjacency list from the first entry to skip all the black vertices and insert into the stack the leftmost grey vertex. Then the repeat the same for this newly inserted vertex into the stack until we reconstruct the last $n/2$ grey vertices. As we have stored the last vertex to be deleted for maintaining the invariant true, we know when to stop this reconstruction procedure. It is not hard to see that this procedure correctly reconstructs the latest set of grey vertices in the stack. We continue this process until all the vertices become black. Moreover, this algorithm runs in $O(m+n)$ time as it involves two phases each taking linear time in the worst case, and uses at most $(n \lg n)/2 + n \lg 3$ bits which fits in our budget of free space in the offsets part of the adjacency array. This completes the description of the linear time in-place DFS algorithm.

Before providing the algorithms for other problems, we need a few additional ideas which we will describe next. In the following theorem, we are interested in dynamically maintaining the degree sequence of all vertices that belong to a spanning subgraph of the original graph. More specifically, given a graph $G = (V, E)$, we want to run some algorithm on G for constructing a sparse spanning subgraph $G' = (V, E')$ (which is a spanning subgraph of G i.e., $E' \subseteq E$ and $|E'| = O(V)$) of G, and we are interested in dynamically maintaining the degree of all the vertices v in G' i.e., degree of a vertex v in G' is defined as the number of neighbors u such that the edge (v, u) belongs to G'. Thus, degree of a vertex v in G' may not be same as degree of v in G. Also note that, by the notion of dynamic, we mean that the algorithm starts with an empty graph and gradually add edges to it before finally culminating with a sparse spanning subgraph, thus during the execution of this algorithm degrees of the individual vertices are changing, and it is this dynamically changing degrees that we want to efficiently maintain. We refer to this as the *dynamic maintenance of degree sequence* phase. Towards this goal, we prove the following general theorem.

Theorem 3. *Given a graph G with n vertices and m edges, let G' be a spanning subgraph of G with m' edges, and also let $d' = m'/n$ be the average degree of G'. Then, we can construct the dynamically created degree sequence for the vertices of G' in $O(m + n)$ time using $O(n(\lg d' + \lg \lg n))$ bits of construction space. Moreover, the final degree sequence can be stored using $O(n \lg d')$ bits such that degree of any vertex can be returned in $O(1)$ time.*

Proof. We divide the vertices into $n/\lg n$ groups of $\lg n$ vertices each. For each group, we allocate a block of $\lg n(\lg d' + \lg \lg n)$ $(\leq \lg^2 n)$ bits initially (uniformly for all the vertices in the block), to store their degrees. We also maintain another parallel bit vector for each block that simply stores the delimiters for each vertex's degree (i.e., a 1 bit to indicate the last bit corresponding to each

vertex's degree, and 0 everywhere else). To access the degree of the i-th vertex in a block, we first find the positions of the $i-1$-th and the i-th 1 bits in the corresponding delimiter sequence, and read the bits between these two positions in the block. To perform this efficiently during the construction, we maintain an auxiliary structure that supports *select* operation in $O(1)$ time [20,38]. At any point, the representation of each block and delimiter sequence consists of an integral number of words, and these representations are maintained as a collection of "extendible arrays" using the structure of [41, Lemma 1].

At any time, a vertex has some number of bits allocated to store its degree. If the degree of the vertex can be updated in-place, then we first access the position where the degree of the vertex is stored, using the select data structure stored for the corresponding delimiter sequence, and update the degree of the node stored within the block. Otherwise, we first note that at least $\lg n$ increments have been performed to some vertex within the block (since each vertex has a 'slack' of $\lg \lg n$ bits at the beginning of the latest re-construction of the block). Now, we spend $O(\lg n)$ time to re-construct the block (and also the corresponding delimiter sequence with its select structure) so that the degree of each vertex v in the block is stored $\lceil \lg d_v \rceil + \lg \lg n$ bits, where d_v is the current degree of v. This $\lg n$ construction time can be amortized over the $\lg n$ increments performed on the block before its re-construction, incurring an $O(1)$ amortized cost per increment. Once we construct the degree sequence for the entire subgraph G', we can scan all the blocks, and compact the degree sequence so that it occupies $O(n \lg d')$ bits. The space usage during the construction is bounded by $O(n(\lg d' + \lg \lg n))$ bits of space. Note that, the above task can be performed while executing the linear time DFS algorithm described before, and this completes the proof.

Corollary 1. *When G' is the DFS tree of G, then we can store the dynamically created degree sequence of G', whose size is bounded by $2n$ bits, by running a linear time DFS procedure while using $O(n \lg \lg n)$ bits of space during construction such that the degree of any vertex in G' can be accessed in $O(1)$ time.*

For the following discussion, assume that we are working with connected undirected graphs only, and given this, now we are going to describe the *setting up parent* phase. More specifically, while performing DFS, suppose we visit the vertex u for the first time from the vertex v (hence v becomes the parent of u in the DFS tree), at that point we perform one or more swaps in the portion of the adjacency array Z where the neighbors of u are located so that the vertex v becomes the first neighbor of u now. If the initial configuration of Z already satisfies this property in u's neighborhood, we don't need to do anything else. We repeat this procedure for every vertex $v \in V$ so that when DFS ends, the first neighbor of every vertex v (except the root vertex) is its parent in the DFS tree. Note that we can perform this step of setting up parent in the first location of every neighborhood list of every vertex alongside performing the linear time DFS algorithm of Theorem 1. Thus, we obtain the following.

Lemma 2. *There exists a linear time in-place algorithm for performing the setting up parent procedure for every vertex of G.*

Note that, by choosing appropriate parameters, we can actually perform the *dynamic maintenance of degree sequence* and the *setting up parent* phase together while running the linear time in-place DFS algorithm of Theorem 1 in any graph G. More specifically, suppose we choose to run the linear time in-place DFS algorithm of Theorem 1 coupled with the setting up parent procedure (to implement Lemma 2) by storing $n/2$ vertices (thus taking $n \lg n/2$ bits) in the free space of the offsets part of Z, thus, leaving roughly $(n \lg n/2 - 2n)$ bits of space still free, which can be used to construct and store the degree sequence of all the vertices in the DFS tree (to implement Corollary 1) while running the same linear time in-place DFS algorithm of Theorem 1. By degree of a vertex v in the DFS tree T, we mean the number of children v has in T, and it is this number that gets stored using the algorithm of Corollary 1. Hence, at the end of this linear time in-place procedure, we have the following invariant: (a) the first neighbor of every vertex (except the root) is its parent in the DFS tree, and (b) the offsets part of Z contains the degree sequence of every vertex v in the DFS tree, and this occupies at most $2n$ bits.

Armed with the above algorithm, we are going to explain next the *implicit representation of the search tree* phase. The goal of this phase is to rearrange the neighbors of any vertex v in such a way that the first neighbor of v becomes its parent in the DFS tree (except for the root vertex), followed by all of v's children in the DFS tree (if any) one by one, finally all the non-child neighbors. Thanks to the setting up parent phase, we can implement the implicit representation the search tree phase in linear time overall by doing a reverse search. More specifically, for every non-root vertex v, we start by scanning v's list from the second neighbor onward (as first neighbor is its parent), and for each one of them, say u, we go to the first location of u's neighbor list to check if v is u's parent if so, we move u in v's list closer to v's parent (i.e., towards the beginning of v's list) by swapping, and repeat this procedure for all the neighbors of v's so that at the end all the children of v are clustered together followed by v's parent. The root vertex can be handled similarly, but we need to start the scanning procedure from the first neighbor itself as it doesn't have any parent. Hence, we spend time proportional to its degree at every vertex, and obtain the following.

Lemma 3. *There exists a linear time in-place algorithm for implicitly representing the search tree of G.*

Thus, from now on we can assume that the neighbor list of every vertex is represented in the search tree format implicitly. We choose to call it so as, note that, given in this format, it is very convenient to answer the following queries for any given vertex v in the DFS tree T: (a) return the parent of v in T in $O(1)$ time, (b) return the number of children v has in T in $O(1)$ time (from the dynamically maintained degree sequence), and finally, (c) enumerate all the children of v one by one optimally in time proportional to its number of children. Not only this, observe that we can still perform the DFS traversal of G optimally in linear-time using essentially the same algorithm of Theorem 1 given this representation. We can even slightly optimize this DFS algorithm by stop scanning the neighbor list of any vertex v as soon as we encounter its last child u in the DFS tree (can be derived from the dynamically maintained degree

sequence) as neighbors after u will not be of significance in performing the DFS traversal of G. Hence, we obtain the following.

Lemma 4. *There exists a linear time in-place algorithm for performing the* DFS *traversal of a given graph G using the implicit search tree representation of G.*

Topological Sorting. One of the standard algorithms for computing topological sort [21] works by simply reporting the vertices of a DFS traversal of a given directed acyclic graph in reverse order. We can easily implement this in-place in linear time by running our DFS algorithm in two phases. More specifically, in the first phase, we run the DFS algorithm completely to generate/store the last $n/2$ vertices in the DFS traversal order, and then report them in reverse order. This is followed by running the DFS algorithm one more time but stopping just when we obtain the other $n/2$ vertices, then we reverse the order of this vertices and report. This completes the description of generating the vertices in topologically sorted order of an input directed acyclic graph in-place in linear time.

In the full version [15] of this paper, we describe linear-time in-place algorithms for all the remaining graph algorithms mentioned in Theorem 1, namely, ordered BFS, MCS, *st*-numbering, MST, chain decomposition, checking biconnectivity and/or 2-edge connectivity, and finding cut vertices and bridges.

4 Conclusions

In this paper, we designed linear time in-place algorithms for a variety of graph problems. As a consequence, many interesting and contrasting observations follow. For example, for *directed st-reachability*, the most space efficient polynomial time algorithm [7] in ROM uses $n/2^{\Theta(\sqrt{\lg n})}$ bits. In sharp contrast, we obtain optimal linear time using logarithmic extra space algorithms for this problem as a simple corollary of both BFS and DFS. Thus, in terms of workspace this is exponentially better than the best known polynomial time algorithm [7] in ROM. This provided us with one of the main motivations for designing algorithms in the *in-place* model. A somewhat incomparable result obtained by Buhrman et al. [9,35] where they gave 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 $O(n^9)$. Finally, we conclude by mentioning that we barely scratched the surface of designing in-place graph algorithms with plenty of more to be studied in this model in future. For example, can we design linear time in-place algorithms for testing planarity of a graph? Can we compute the max-flow/min-cut in-place? Can we compute shortest paths between any two vertices of a given graph in-place? We leave these problems as our future directions of study.

References

1. Alon, N., Matias, Y., Szegedy, M.: The space complexity of approximating the frequency moments. J. Comput. Syst. Sci. **58**(1), 137–147 (1999)
2. Asano, T., et al.: Reprint of: memory-constrained algorithms for simple polygons. Comput. Geom. **47**(3), 469–479 (2014)

3. Asano, T., et al.: Depth-first search using $O(n)$ bits. In: Ahn, H.-K., Shin, C.-S. (eds.) ISAAC 2014. LNCS, vol. 8889, pp. 553–564. Springer, Cham (2014). https://doi.org/10.1007/978-3-319-13075-0_44
4. Asano, T., Mulzer, W., Rote, G., Wang, Y.: Constant-work-space algorithms for geometric problems. JoCG **2**(1), 46–68 (2011)
5. Banerjee, N., Chakraborty, S., Raman, V., Satti, S.R.: Space efficient linear time algorithms for BFS, DFS and applications. Theory Comput. Syst. **62**(8), 1736–1762 (2018)
6. Barba, L., Korman, M., Langerman, S., Sadakane, K., Silveira, R.I.: Space-time trade-offs for stack-based algorithms. Algorithmica **72**(4), 1097–1129 (2015)
7. Barnes, G., Buss, J., Ruzzo, W., Schieber, B.: A sublinear space, polynomial time algorithm for directed *s*-*t* connectivity. SIAM J. Comput. **27**(5), 1273–1282 (1998)
8. Brönnimann, H., Chan, T.M., Chen, E.Y.: Towards in-place geometric algorithms and data structures. In: SOCG, pp. 239–246 (2004)
9. Buhrman, H., Cleve, R., Koucký, M., Loff, B., Speelman, F.: Computing with a full memory: catalytic space. In: STOC, pp. 857–866 (2014)
10. Chakraborty, S.: Space efficient graph algorithms. Ph.D. thesis, The Institute of Mathematical Sciences, HBNI, India (2018)
11. Chakraborty, S., Mukherjee, A., Raman, V., Satti, S.R.: A framework for in-place graph algorithms. In: ESA, pp. 13:1–13:16 (2018)
12. Chakraborty, S., Mukherjee, A., Satti, S.R.: Space efficient algorithms for breadth-depth search. In: Gąsieniec, L.A., Jansson, J., Levcopoulos, C. (eds.) FCT 2019. LNCS, vol. 11651, pp. 201–212. Springer, Cham (2019). https://doi.org/10.1007/978-3-030-25027-0_14
13. Chakraborty, S., Raman, V., Satti, S.R.: Biconnectivity, st-numbering and other applications of DFS using O(n) bits. J. Comput. Syst. Sci. **90**, 63–79 (2017)
14. Chakraborty, S., Sadakane, K.: Indexing graph search trees and applications. In: 44th MFCS. LIPIcs, vol. 138, pp. 67:1–67:14. Schloss Dagstuhl - Leibniz-Zentrum für Informatik (2019)
15. Chakraborty, S., Sadakane, K., Satti, S.R.: Optimal in-place algorithms for basic graph problems. CoRR, abs/1907.09280 (2019)
16. Chakraborty, S., Satti, S.R.: Space-efficient algorithms for maximum cardinality search, its applications, and variants of BFS. J. Comb. Optim. **37**(2), 465–481 (2018)
17. Chan, T.M., Chen, E.Y.: Multi-pass geometric algorithms. Discret. Comput. Geom. **37**(1), 79–102 (2007)
18. Chan, T.M., Munro, J.I., Raman, V.: Faster, space-efficient selection algorithms in read-only memory for integers. In: Cai, L., Cheng, S.-W., Lam, T.-W. (eds.) ISAAC 2013. LNCS, vol. 8283, pp. 405–412. Springer, Heidelberg (2013). https://doi.org/10.1007/978-3-642-45030-3_38
19. Chan, T.M., Munro, J.I., Raman, V.: Selection and sorting in the "restore" model. ACM Trans. Algorithms **14**(2), 11:1–11:18 (2018)
20. Clark, D.R.: Compact pat trees. Ph.D. thesis. University of Waterloo, Canada (1996)
21. Cormen, T.H., Leiserson, C.E., Rivest, R.L., Stein, C.: Introduction to Algorithms, 3rd edn. MIT Press, Cambridge (2009)
22. Darwish, O., Elmasry, A.: Optimal time-space tradeoff for the 2D convex-hull problem. In: Schulz, A.S., Wagner, D. (eds.) ESA 2014. LNCS, vol. 8737, pp. 284–295. Springer, Heidelberg (2014). https://doi.org/10.1007/978-3-662-44777-2_24
23. Diestel, R.: Graph Theory, 4th edn (2012)

24. Elias, P.: Efficient storage and retrieval by content and address of static files. J. ACM **21**(2), 246–260 (1974)
25. Elmasry, A., Hagerup, T., Kammer, F.: Space-efficient basic graph algorithms. In: 32nd STACS, pp. 288–301 (2015)
26. Elmasry, A., Juhl, D.D., Katajainen, J., Satti, S.R.: Selection from read-only memory with limited workspace. Theor. Comput. Sci. **554**, 64–73 (2014)
27. Even, S., Tarjan, R.E.: Computing an *st*-numbering. Theor. Comput. Sci. **2**(3), 339–344 (1976)
28. Fano, R.M.: On the number of bits required to implement an associative memory. Memorandum 61, Computer Structures Group, MIT, Cambridge (1971)
29. Feigenbaum, J., Kannan, S., McGregor, A., Suri, S., Zhang, J.: On graph problems in a semi-streaming model. Theor. Comput. Sci. **348**(2–3), 207–216 (2005)
30. Franceschini, G., Munro, J.I.: Implicit dictionaries with $O(1)$ modifications per update and fast search. In: SODA, pp. 404–413 (2006)
31. Franceschini, G., Muthukrishnan, S.: In-place suffix sorting. In: Arge, L., Cachin, C., Jurdziński, T., Tarlecki, A. (eds.) ICALP 2007. LNCS, vol. 4596, pp. 533–545. Springer, Heidelberg (2007). https://doi.org/10.1007/978-3-540-73420-8_47
32. Franceschini, G., Muthukrishnan, S., Pătraşcu, M.: Radix sorting with no extra space. In: Arge, L., Hoffmann, M., Welzl, E. (eds.) ESA 2007. LNCS, vol. 4698, pp. 194–205. Springer, Heidelberg (2007). https://doi.org/10.1007/978-3-540-75520-3_19
33. Frederickson, G.N.: Upper bounds for time-space trade-offs in sorting and selection. J. Comput. Syst. Sci. **34**(1), 19–26 (1987)
34. Kammer, F., Sajenko, A.: Linear-time in-place DFS and BFS on the word RAM. In: Heggernes, P. (ed.) CIAC 2019. LNCS, vol. 11485, pp. 286–298. Springer, Cham (2019). https://doi.org/10.1007/978-3-030-17402-6_24
35. Koucký, M.: Catalytic computation. Bull. EATCS **118** (2016). http://eatcs.org/beatcs/index.php/beatcs/article/view/400
36. Lai, T.W., Wood, D.: Implicit selection. In: Karlsson, R., Lingas, A. (eds.) SWAT 1988. LNCS, vol. 318, pp. 14–23. Springer, Heidelberg (1988). https://doi.org/10.1007/3-540-19487-8_2
37. Munro, J.I.: An implicit data structure supporting insertion, deletion, and search in O($\log^2$ n) time. J. Comput. Syst. Sci. **33**(1), 66–74 (1986)
38. Munro, J.I.: Tables. In: FSTTCS, pp. 37–42 (1996)
39. Munro, J.I., Paterson, M.: Selection and sorting with limited storage. Theor. Comput. Sci. **12**, 315–323 (1980)
40. Pagter, J., Rauhe, T.: Optimal time-space trade-offs for sorting. In: FOCS, pp. 264–268 (1998)
41. Raman, R., Rao, S.S.: Succinct dynamic dictionaries and trees. In: Baeten, J.C.M., Lenstra, J.K., Parrow, J., Woeginger, G.J. (eds.) ICALP 2003. LNCS, vol. 2719, pp. 357–368. Springer, Heidelberg (2003). https://doi.org/10.1007/3-540-45061-0_30
42. Schmidt, J.M.: A simple test on 2-vertex- and 2-edge-connectivity. Inf. Process. Lett. **113**(7), 241–244 (2013)
43. Tarjan, R.E.: Depth-first search and linear graph algorithms. SIAM J. Comput. **1**(2), 146–160 (1972)
44. Tarjan, R.E.: A note on finding the bridges of a graph. Inf. Process. Lett. **2**(6), 160–161 (1974)
45. Tarjan, R.E., Yannakakis, M.: Simple linear-time algorithms to test chordality of graphs, test acyclicity of hypergraphs, and selectively reduce acyclic hypergraphs. SIAM J. Comput. **13**(3), 566–579 (1984)