Depth First Search in the Semi-streaming Model

Shahbaz Khan
Faculty of Computer Science, University of Vienna, Austria
shahbaz.khan@univie.ac.at

Shashank K. Mehta
Dept. of Computer Science and Engineering, Indian Institute of Technology Kanpur, India
skmehta@cse.iitk.ac.in

Abstract

Depth first search (DFS) tree is a fundamental data structure for solving various graph problems. The classical algorithm for building a DFS tree requires $O(m+n)$ time for a given undirected graph G having n vertices and m edges. In the streaming model, an algorithm is allowed several passes (preferably single) over the input graph having a restriction on the size of local space used.

Now, a DFS tree of a graph can be trivially computed using a single pass if $O(m)$ space is allowed. In the semi-streaming model allowing $O(n)$ space, it can be computed in $O(n)$ passes over the input stream, where each pass adds one vertex to the DFS tree. However, it remains an open problem to compute a DFS tree using $o(n)$ passes using $o(m)$ space even in any relaxed streaming environment.

We present the first semi-streaming algorithms that compute a DFS tree of an undirected graph in $o(n)$ passes using $o(m)$ space. We first describe an extremely simple algorithm that requires at most $\lceil n/k \rceil$ passes to compute a DFS tree using $O(nk)$ space, where k is any positive integer. For example using $k = \sqrt{n}$, we can compute a DFS tree in $\sqrt{n}$ passes using $O(n\sqrt{n})$ space. We then improve this algorithm by using more involved techniques to reduce the number of passes to $\lceil h/k \rceil$ under similar space constraints, where h is the height of the computed DFS tree. In particular, this algorithm improves the bounds for the case where the computed DFS tree is *shallow* (having $o(n)$ height). Moreover, this algorithm is presented in form of a *framework* that allows the *flexibility* of using any algorithm to maintain a DFS tree of a stored sparser subgraph as a *black box*, which may be of an independent interest. Both these algorithms essentially demonstrate the existence of a trade-off between the space and number of passes required for computing a DFS tree. Furthermore, we evaluate these algorithms experimentally which reveals their exceptional performance in practice. For both random and real graphs, they require merely a *few* passes even when allowed just $O(n)$ space.

2012 ACM Subject Classification Theory of computation → Graph algorithms analysis; Theory of computation → Data structures design and analysis; Theory of computation → Streaming, sublinear and near linear time algorithms

Keywords and phrases Depth First Search, DFS, Semi-Streaming, Streaming, Algorithm

Digital Object Identifier 10.4230/LIPIcs.STACS.2019.42

Related Version Full version of the papers is available at `https://arxiv.org/abs/1901.03689`.

Funding *Shahbaz Khan*: This research work was supported by the European Research Council under the European Union's Seventh Framework Programme (FP/2007-2013) / ERC Grant Agreement no. 340506.

1 Introduction

Depth first search (DFS) is a well known graph traversal technique. Right from the seminal work of Tarjan [35], DFS traversal has played an important role in the design of efficient algorithms for many fundamental graph problems, namely, bi-connected components, strongly connected components, topological sorting [38], dominators in directed graph [36], etc. Even in undirected graphs, DFS traversal have various applications including computing connected components, cycle detection, edge and vertex connectivity [13] (via articulation points and bridges), bipartite matching [23], planarity testing [24] etc. In this paper, we address the problem of computing a DFS tree in the semi-streaming environment.

The streaming model [2, 17, 19] is a popular model for computation on large data sets wherein a lot of algorithms have been developed [18, 22, 19, 25] to address significant problems in this model. The model requires the entire input data to be accessed as a stream, typically in a single pass over the input, allowing very small amount of storage ($poly \log$ in input size). A streaming algorithm must judiciously choose the data to be saved in the small space, so that the computation can be completed successfully. In the context of graph problems, this model is adopted in the following fashion. For a given graph $G = (V, E)$ having n vertices, an input stream sends the graph edges in E using an arbitrary order only once, and the allowed size of local storage is $O(poly \log n)$. The algorithm iteratively asks for the next edge and performs some computation. After the stream is over, the final computation is performed and the result is reported. At no time during the entire process should the total size of stored data exceed $O(poly \log n)$.

In general only statistical properties of the graph are computable under this model, making it impractical for use in more complicated graph problems [15, 20]. A prominent exception for the above claim is the problem of counting triangles (3-cycles) in a graph [5]. Consequently, several relaxed models have been proposed with a goal to solve more complex graph problems. One such model is called *semi-streaming model* [32, 16] which relaxes the storage size to $O(n\ poly \log n)$. Several significant problems have been studied under this model (surveys in [33, 41, 31]). Moreover, even though it is preferred to allow only a single pass over the input stream, several hardness results [22, 10, 16, 9, 21] have reported the limitations of using a single pass (or even $O(1)$ passes). This has led to the development of various multi-pass algorithms [16, 15, 30, 1, 27, 26] in this model. Further, several streaming algorithms maintaining approximate distances [15, 6, 11] are also known to require $O(n^{1+\epsilon})$ space (for some constant $\epsilon > 0$) relaxing the requirement of $O(n\ poly \log n)$ space.

Now, a DFS tree of a graph can be computed in a single pass if $O(m)$ space is allowed. If the space is restricted to $O(n)$, it can be trivially computed using $O(n)$ passes over the input stream, where each pass adds one vertex to the tree. This can also be easily improved to $O(h)$ passes, where h is the height of the computed DFS tree. Despite most applications of DFS trees in undirected graphs being efficiently solved in the semi-streaming environment [40, 16, 15, 3, 4, 14, 29], due to its fundamental nature DFS is considered a long standing open problem [14, 33, 34] even for undirected graphs. Moreover, computing a DFS tree in $O(poly \log n)$ passes is considered hard [14]. To the best of our knowledge, it remains an open problem to compute a DFS tree using $o(n)$ passes even in any relaxed streaming environment.

In our results, we borrow some key ideas from recent sequential algorithms [8, 7] for maintaining dynamic DFS of undirected graphs. Recently, similar ideas were also used by Khan [28] who presented a semi-streaming algorithm that uses using $O(n)$ space for maintaining dynamic DFS of an undirected graph, requiring $O(\log^2 n)$ passes per update.

1.1 Our Results

We present the first semi-streaming algorithms to compute a DFS tree on an undirected graph in $o(n)$ passes. Our first result can be described using the following theorem.

▶ **Theorem 1.** *Given an undirected graph $G = (V, E)$, the DFS tree of the graph can be computed by a semi-streaming algorithm in at most n/k passes using $O(nk)$ space, requiring $O(m\alpha(m, n))$ time per pass.*

As described earlier, a simple algorithm can compute the DFS tree in $O(h)$ passes, where h is the height of the DFS tree. Thus, for the graphs having a DFS tree with height $h = o(n)$ (see full paper for details), we improve our result for such graphs in the following theorem.

▶ **Theorem 2.** *Given an undirected graph G, a DFS tree of G can be computed by a semi-streaming algorithm using $\lceil h/k \rceil$ passes using $O(nk)$ space requiring amortized $O(m + nk)$ time per pass for any integer $k \leq h$, where h is the height of the computed DFS tree.*[1]

Since typically the space allowed in the semi-streaming model is $O(n\ poly \log n)$, the improvement in upper bounds of the problem by our results is considerably small (upto $poly \log n$ factors). Recently, Elkin [12] presented the first $o(n)$ pass algorithm for computing Shortest Paths Trees. Using $O(nk)$ local space, it computes the shortest path tree from a given source in $O(n/k)$ passes for unweighted graphs, and in $O(n \log n/k)$ passes for weighted graphs. The significance of such results, despite improving the upper bounds by only small factors, is substantial because they address fundamental problems. The lack of any progress for such fundamental problems despite several decades of research on streaming algorithms further highlights the significance of such results. Moreover, allowing $O(n^{1+\epsilon})$ space (as in [15, 6, 11]) such results improves the upper bound significantly by $O(n^{\epsilon})$ factors. Furthermore, they demonstrate the existence of a trade-off between the space and number of passes required for computing such fundamental structures.

Our final algorithm is presented in form of a *framework*, which can use any algorithm for maintaining a DFS tree of a stored sparser subgraph, provided that it satisfies the property of *monotonic fall*. Such a *framework* allows more flexibility and is hopefully much easier to extend to better algorithms for computing a DFS tree or other problems requiring a computation of DFS tree. Hence we believe our *framework* would be of independent interest.

We also augment our theoretical analysis with the experimental evaluation of our proposed algorithms. For both random and real graphs, the algorithms require merely a *few* passes even when the allowed space is just $O(n)$. The exceptional performance and surprising observations of our experiments on random graphs might also be of independent interest.

1.2 Overview

We now briefly describe the outline of our paper. In Section 2 we establish the terminology and notations used in the remainder of the paper. In order to present the main ideas behind our approach in a simple and comprehensible manner, we present the algorithm in four stages. *Firstly* in Section 3, we describe the basic algorithm to build a DFS tree in n passes, which adds a new vertex to the DFS tree in every pass over the input stream. *Secondly* in Section 3.1, we improve this algorithm to compute a DFS tree in h passes, where h is the height of the final DFS tree. This algorithm essentially computes all the vertices in the next level of the currently built DFS tree simultaneously, building the DFS tree by one level in each pass over the input stream. Thus, in the i^{th} pass every vertex on the i^{th} level of the DFS tree is computed. *Thirdly* in Section 4, we describe an advanced algorithm which uses

[1] Note that there can be many DFS trees of a graph having varying heights, say h_{min} to h_{max}. Our algorithm does not guarantee the computation of DFS tree having minimum height h_{min}, rather it simply computes a *valid* DFS tree with height h, where $h_{min} \leq h \leq h_{max}$.

$O(nk)$ space to add a path of length at least k to the DFS tree in every pass over the input stream. Thus, the complete DFS tree can be computed in $\lceil n/k \rceil$ passes. *Finally*, in Section 5, we improve the algorithm to simultaneously add all the subtrees constituting the next k levels of the final DFS tree starting from the leaves of the current tree T. Thus, k levels are added to the DFS tree in each pass over the input stream, computing the DFS tree in $\lceil h/k \rceil$ passes. As described earlier, our final algorithm is presented in form of a *framework* which uses as a black box, any algorithm to maintain a DFS tree of a stored sparser subgraph, satisfying certain properties. In the interest of completeness, one such algorithm is described in the full paper. *Lastly* in Section 6, we present the results of the experimental evaluation of these algorithms. The details of this evaluation are deferred to the full version of the paper.

In our advanced algorithms, we employ two interesting properties of a DFS tree, namely, the *components* property [7] and the *min-height* property. These simple properties of any DFS tree prove crucial in building the DFS efficiently in the streaming environment.

2 Preliminaries

Let $G = (V, E)$ be an undirected connected graph having n vertices and m edges. The DFS traversal of G starting from any vertex $r \in V$ produces a spanning tree rooted at r called a DFS tree, in $O(m + n)$ time. For any rooted spanning tree of G, a non-tree edge of the graph is called a *back edge* if one of its endpoints is an ancestor of the other in the tree, else it is called a *cross edge*. A necessary and sufficient condition for any rooted spanning tree to be a DFS tree is that every non-tree edge is a back edge.

In order to handle disconnected graphs, we add a dummy vertex r to the graph and connect it to all vertices. Our algorithm computes a DFS tree rooted at r in this augmented graph, where each child subtree of r is a DFS tree of a connected component in the DFS forest of the original graph. The following notations will be used throughout the paper.

- T : The DFS tree of G incrementally computed by our algorithm.
- $par(w)$: Parent of w in T.
- $T(x)$: The subtree of T rooted at vertex x.
- $root(T')$: Root of a subtree T' of T, i.e., $root\big(T(x)\big) = x$.
- $level(v)$: Level of vertex v in T, where $level(root(T)) = 0$ and $level(v) = level(par(v))+1$.

In this paper we will discuss algorithms to compute a DFS tree T for the input graph G in the semi-streaming model. In all the cases T will be built iteratively starting from an empty tree. At any time during the algorithm, we shall refer to the vertices that are not a part of the DFS tree T as *unvisited* and denote them by V', i.e., $V' = V \setminus T$. Similarly, we refer to the subgraph induced by the *unvisited* vertices, $G' = G(V')$, as the *unvisited graph*. Unless stated otherwise, we shall refer to a connected component of the unvisited graph G' as simply a *component*. For any component C, the set of edges and vertices in the component will be denoted by E_C and V_C. Further, each component C maintains a spanning tree of the component that shall be referred as T_C. We refer to a path p in a DFS tree T as an *ancestor-descendant* path if one of its endpoints is an ancestor of the other in T. Since the DFS tree grows downwards from the root, a vertex u is said to be *higher* than vertex v if $level(u) < level(v)$. Similarly, among two edges incident on an ancestor-descendant path p, an edge (x, y) is *higher* than edge (u, v) if $y, v \in p$ and $level(y) < level(v)$.

We shall now describe two invariants such that any algorithm computing DFS tree incrementally satisfying these invariants at every stage of the algorithm, ensures the absence of cross edges in T and hence the correctness of the final DFS tree T.

Invariants:

$\mathcal{I}_1$: All non-tree edges among vertices in T are back edges, and
$\mathcal{I}_2$: For any component C of the unvisited graph, all the edges from C to the partially built DFS tree T are incident on a single *ancestor-descendant* path of T.

We shall also use the *components property* by Baswana et al. [7], described as follows.

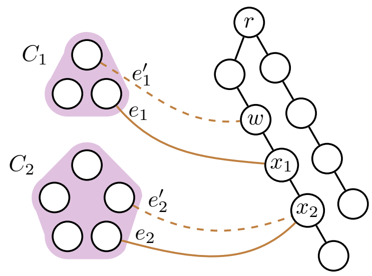

Figure 1 Edges e'_1 and e'_2 can be ignored during the DFS traversal (reproduced from [7]).

▶ **Lemma 3** (Components Property [7]). *Consider a partially completed DFS traversal where T is the partially built DFS tree. Let the connected components of G' be $C_1, .., C_k$. Consider any two edges e_i and e'_i from C_i that are incident respectively on a vertex x_i and some ancestor (not necessarily proper) w of x_i in T. Then it is sufficient to consider only e_i during the DFS traversal, i.e., the edge e'_i can be safely ignored.*

Ignoring e'_i during the DFS traversal, as stated in the components property, is justified because e'_i will appear as a back edge in the resulting DFS tree (refer to Figure 1). For each component C_i, the edge e_i can be found using a single pass over all the graph edges.

3 Simple Algorithms

We shall first briefly describe the trivial algorithm to compute a DFS tree of a (directed) graph using n passes. Since we are limited to have only $O(n \, poly \log n)$ space, we cannot store the adjacency list of the vertices in the graph. Recall that in the standard DFS algorithm [35], after visiting a vertex v, we choose any unvisited neighbour of v and visit it. If no neighbour of v is unvisited, the traversal retreats back to the parent of v and look for its unvisited neighbour, and so on.

In the streaming model, we can use the same algorithm. However, we do not store the adjacency list of a vertex. To find the unvisited neighbour of each vertex, we perform a complete pass over the edges in E. The algorithm only stores the partially built DFS tree and the status of each vertex (whether it is visited/added to T). Thus, for each vertex v (except r) one pass is performed to add v to T and another is performed before retreating to the parent of v. Hence, it takes $2(n-1)$ passes to complete the algorithm since T is initialized with the root r. Since, this procedure essentially simulates the standard DFS algorithm [35], it clearly satisfies the invariants $\mathcal{I}_1$ and $\mathcal{I}_2$.

This procedure can be easily transformed to require only $n-1$ passes by avoiding an extra pass for retreating from each vertex v. In each pass we find an edge e (from the stream) from the unvisited vertices, V', to the lowest vertex on the ancestor-descendant path connecting r and v, i.e., closest to v. Hence e would be an edge from the lowest (maximum level) ancestor of v (not necessarily proper) having at least one unvisited neighbour. Recall that if v does not have an unvisited neighbour we move to processing its parent, and so on until we find an ancestor having an unvisited neighbour. We can thus directly add the edge e to T. Hence, retreating from a vertex would not require an additional pass and the overall procedure can be completed in $n-1$ passes, each pass adding a new vertex to T. Moreover, this also requires $O(1)$ processing time per edge and extra $O(n)$ time at the end of the pass, to find the relevant ancestor. Refer to the full paper for the pseudocode of the procedure. Thus, we get the following result.

▶ **Theorem 4.** *Given a directed/undirected graph G, a DFS tree of G can be computed by a semi-streaming algorithm in n passes using $O(n)$ space, using $O(m)$ time per pass.*

3.1 Improved algorithm

We shall now describe how this simple algorithm can be improved to compute a DFS tree of an undirected graph in h passes, where h is the height of the computed DFS tree. The main idea behind this approach is that each component of the unvisited graph G' will constitute a separate subtree of the final DFS tree. Hence each such subtree can be computed independent of each other in parallel (this idea was also used by [28]).

Using one pass over edges in E, the components of the unvisited graph G' can be found by using Union-Find algorithm [37, 39] on the edges E' of G'. Now, using the *components* property we know that it is sufficient to add the lowest edge from each component to the DFS tree T. At the end of the pass, for each component C we find the edge (x_C, y_C) incident from the lowest vertex $x_C \in T$ to some vertex $y_C \in V_C$ and add it to T. Note that in the next pass, for each component of $C \setminus \{y_C\}$ the lowest edge connecting it to T would necessarily be incident on y_C as C was connected. Hence, instead of lowest edge incident on T, we store e_y from $y \in V'$ only if e_y is incident on some leaf of T. Refer to the full paper for the pseudocode of the algorithm.

To prove the correctness of the algorithm, we shall prove using induction that the invariants $\mathcal{I}_1$ and $\mathcal{I}_2$ hold over the passes performed on E. Since T is initialized as an isolated vertex r, both invariants trivially hold. Now, let the invariants hold at the beginning of a pass. Using $\mathcal{I}_2$, each component C can have edges to a single ancestor-descendant path from r to x_C. Thus, adding the edge (x_C, y_C) for each component C, would not violate $\mathcal{I}_1$ at the end of the pass, given that $\mathcal{I}_1$ holds at the beginning of the pass. Additionally, from each component C we add a single vertex y_C as a child of x_C to T. Hence for any component of $C \setminus \{y_C\}$, the edges to T can only be to ancestors of y_C (using $\mathcal{I}_2$ of previous pass), and an edge necessarily to y_C, satisfying $\mathcal{I}_2$ at the end of the pass. Hence, using induction both $\mathcal{I}_1$ and $\mathcal{I}_2$ are satisfied proving the correctness of our algorithm.

Further, since each component C in any i^{th} pass necessarily has an edge to a leaf x_C of T, the new vertex y_C is added to the i^{th} level of T. This also implies that every vertex at i^{th} level of the final DFS tree is added during the i^{th} pass. Hence, after h passes we get a DFS tree of the whole graph as h is the height of the computed DFS tree.

Now, the total time[2] required to compute the connected components is $O(m\alpha(m,n))$. And computing an edge from each unvisited vertex to a leaf in T requires $O(1)$ time using $O(n)$ space. Thus, we have the following theorem.

[2] The Union-Find algorithm [37, 39] requires $O(m\alpha(m,n))$ time, where $\alpha(m,n)$ is the inverse Ackermann function.

▶ **Theorem 5.** *Given an undirected graph G, a DFS tree of G can be computed by a semi-streaming algorithm in h passes using $O(n)$ space, where h is the height of the computed DFS tree, using $O(m\alpha(m,n))$ time per pass.*

4 Computing DFS in sublinear number of passes

Since a DFS tree may have $O(n)$ height, we cannot hope to compute a DFS tree in sublinear number of passes using the previously described simple algorithms. The main difference between the advanced approaches and the simple algorithms is that, in each pass instead of adding a single vertex (say y) to the DFS tree, we shall be adding an entire path (starting from y) to the DFS tree. The DFS traversal gives the flexibility to chose the next vertex to be visited as long as the DFS property is satisfied, i.e., invariants $\mathcal{I}_1$ and $\mathcal{I}_2$ are maintained.

Hence in each pass we do the following for every component C in G'. Instead of finding a single edge (x_C, y_C) (see Section 3.1), we find a path P starting from y_C in C and attach this entire path P to T (instead of only y_C). Suppose this splits the component C into components $C_1, C_2, \dots$ of $C \setminus P$. Now, each C_i would have an edge to at least one vertex on P (instead of necessarily the leaf x_C in Section 3.1) since C was a connected component. Hence in this algorithm for each C_i, we find the vertex y_i which is the lowest vertex of T (or P) to which an edge from C_i is incident. Observe that y_i is unique since all the neighbours of C_i in T are along one path from the root to a leaf. Using the *components* property, the selection of y_i as the parent of the root of the subtree to be computed for C_i ensures that invariant $\mathcal{I}_2$ continues to hold. Thus, in each pass from every component of the unvisited graph, we shall extract a path and add it to the DFS tree T.

This approach thus allows T to grow by more than one vertex in each pass which is essential for completing the tree in $o(n)$ passes. If in each pass we add a path of length at least k from each component of G', then the tree will grow by at least k vertices in each pass, requiring overall $\lceil n/k \rceil$ passes to completely build the DFS tree. We shall now present an important property of any DFS tree of an undirected graph, which ensures that in each pass we can find a path of length at least $k \geq m/n$.

▶ **Lemma 6** (Min-Height Property). *Given a connected undirected graph G having m edges, any DFS tree of G from any root vertex necessarily has a height $h \geq m/n$.*

Proof. We know that each non-tree edge in a DFS tree of an undirected graph is a *back edge*. We shall associate each edge to its lower endpoint. Thus, in a DFS tree each vertex will be associated to a tree edge to its parent and back edges only to its ancestors. Now, each vertex can have only h ancestors as the height of the DFS tree is h, Hence each vertex has only h edges associated to it resulting in less than nh edges, i.e. $m \leq nh$ or $h \geq m/n$. Note that it is important for the graph to be connected otherwise from some root the corresponding component and hence its DFS tree can be much smaller. ◀

4.1 Algorithm

We shall now describe our algorithm to compute a DFS tree of the input graph in $o(n)$ passes. Let the maximum space allowed for computation in the semi-streaming model be $O(nk)$. The algorithm is a recursive procedure that computes a DFS tree of a component C from a root r_C. For each component C we maintain a spanning tree T_C of C. Initially we can perform a single pass over E to compute a spanning tree of the connected graph G (recall the assumption in Section 2) using the Union-Find algorithm. For the remaining components, its spanning tree would already have been computed and passed as an input to the algorithm.

We initiate a pass over the edge in E and store the first $|V_C| \cdot k$ edges (if possible) from the component C in the memory as the subgraph E'_C. Before proceeding with the remaining stream, we use any algorithm for computing a DFS tree T'_C rooted at r_C in the subgraph containing edges from T_C and E'_C. Note that adding T_C to E'_C is important to ensure that subgraph induced by $T_C \cup E'_C$ is connected. In case the pass was completed before E'_C exceeded storing $|V_C| \cdot k$ edges, T'_C is indeed a DFS tree of C and we directly add it to T. Otherwise, we find the longest path P from T'_C starting from r_C, i.e., path from r_C to the farthest leaf. The path P is then added to T.

Now, we need to compute the connected components of $C \setminus P$ and the new corresponding root for each such component. We use the Union-Find algorithm to compute these components, say $C_1, ..., C_f$, and compute the lowest edge e_i from each C_i on the path P. Clearly, there exist such an edge as C was connected. In order to find these components and edges, we need to consider all the edges in E_C, which can be done by first considering E'_C and then each edge from C in the remainder of input stream of the pass. Refer to the full paper for the pseudocode of the algorithm.

Using the components property, choosing the new root y_i corresponding to the lowest edge e_i ensures that the invariant $\mathcal{I}_2$ and hence $\mathcal{I}_1$ is satisfied. Now, in case $|E_C| < |V_C| \cdot k$, the entire DFS tree of C is constructed and added to T in a single pass. Otherwise, in each pass we add the longest path P from T'_C to the final DFS tree T. Since $|E'_C| = |V_C| \cdot k$ and $E'_C \cup T_C$ is a single connected component, the *min-height property* ensures that the height of any such T'_C (and hence P) is at least k. Since in each pass, except the last, we add at least k new vertices to T, this algorithm terminates in at most $\lceil n/k \rceil$ passes. Now, the total time required to find the components of the unvisited graph is again $O(m\alpha(m, n))$. The remaining operations clearly require $O(|E_C|)$ time for a component C, requiring overall $O(m)$ time. Thus, we get the following theorem.

▶ **Theorem 1.** *Given an undirected graph G, a DFS tree of G can be computed by a semi-streaming algorithm in at most $\lceil n/k \rceil$ passes using $O(nk)$ space, requiring $O(m\alpha(m, n))$ time per pass.*

▶ Remark 7. Since, the algorithm adds an ancestor-descendant path for each component of G', it might seem that the analysis of the algorithm is not tight for computing DFS trees with $o(n)$ height. However, there exist a sequence of input edges where the algorithm indeed takes $\Theta(n/k)$ passes for computing a DFS tree with height $o(n)$. The details of the sequence are described in the full version of the paper.

5 Final algorithm

We shall now further improve the algorithm so that the required number of passes reduces to $\lceil h/k \rceil$ while it continues to use $O(nk)$ space, where h is the height of the computed DFS tree and k is any positive integer. To understand the main intuition behind our approach, let us recall the previously described algorithms. We first described a simple algorithm (in Section 3) in which every pass over the input stream adds one new vertex as the child of some leaf of T, which was improved (in Section 3.1) to simultaneously adding all vertices which are children of the leaves of T in the final DFS tree. We then presented another algorithm (in Section 4) in which every pass over the input stream adds one *ancestor-descendant* path of length k or more, from each component of G' to T. We shall now improve it by adding all the subtrees constituting the next k levels of the final DFS tree starting from the leaves of the current tree T (or fewer than k levels if the corresponding component of G' is exhausted).

Now, consider any component C of G'. Let $r_C \in C$ be a vertex having an edge e to a leaf of the partially built DFS tree T. The computation of T can be completed by computing a DFS tree of C from the root r_C, which can be directly attached to T using e. However, computing the entire DFS tree of C may not be possible in a single pass over the input stream, due to the limited storage space available. Thus, using $O(n \cdot k)$ space we compute a special spanning tree T_C for each component C of G' in parallel, such that the top k levels of T_C is same as the top k levels of *some* DFS tree of C. As a result, in the i^{th} pass all vertices on the levels $(i-1) \cdot k + 1$ to $i \cdot k$ of the final DFS tree are added to T. This essentially adds a tree T'_C representing the top k levels of T_C for each component C of G'. This ensures that our algorithm will terminate in $\lceil h/k \rceil$ passes, where h is the height of the final DFS tree. Further, this special tree T_C also ensures an *additional* property, i.e., there is a one to one correspondence between the set of trees of $T_C \setminus T'_C$ and the components of $C \setminus T'_C$. In fact, each tree of $T_C \setminus T'_C$ is a spanning tree of the corresponding component. This property directly provides the spanning trees of the components of G' in the next pass.

Special spanning tree T_C

We shall now describe the properties of this special tree T_C (and hence T'_C) which is computed in a single pass over the input stream. For T'_C to be added to the DFS tree T of the graph, a necessary and sufficient condition is that T'_C satisfies the invariants $\mathcal{I}_1$ and $\mathcal{I}_2$ at the end of the pass. To achieve this we maintain T_C to be a spanning tree of C, such that these invariants are maintained by the corresponding T'_C throughout the pass as the edges are processed. Let S_C be the set of edges already visited during the current pass, which have both endpoints in C. In order to satisfy $\mathcal{I}_1$, no edge in S_C should be a cross edge in T'_C, i.e., no edge having both endpoints in the top k levels of T_C is a cross edge. In order to satisfy $\mathcal{I}_2$, no edge in S_C from any component $C' \in C \setminus T'_C$ to $C \setminus C'$ should be a cross edge in T_C. Hence, using the *additional* property of T_C, each edge from a tree τ in $T_C \setminus T'_C$ to $T_C \setminus \tau$ is necessarily a back edge. This is captured by the two conditions of invariant $\mathcal{I}_T$ given below. Hence $\mathcal{I}_T$ should hold after processing each edge in the pass. Observe that any spanning tree, T_C, trivially satisfies $\mathcal{I}_T$ at the beginning of the pass as $S_C = \emptyset$.

Invariant $\mathcal{I}_T$:
T_C is a spanning tree of C with the top k levels being T'_C such that:
$\mathcal{I}_{T_1}$: All non-tree edges of S_C having both endpoints in T'_C, are back edges.
$\mathcal{I}_{T_2}$: For each tree τ in $T_C \setminus T'_C$, all the edges of S_C from τ to $T_C \setminus \tau$ are back edges.

Thus, $\mathcal{I}_T$ is the local invariant maintained by T_C during the pass, so that the global invariants $\mathcal{I}_1$ and $\mathcal{I}_2$ are maintained throughout the algorithm. Now, in order to compute T_C (and hence T'_C) satisfying the above invariant, we store a subset of S_C along with T_C. Let H_C denote the (spanning) subgraph of G formed by T_C along with these additional edges. Note that all the edges of S_C cannot be stored in H_C due to space limitation of $O(nk)$. Since each pass starts with the spanning tree T_C of C and $S_C = \emptyset$, initially $H_C = T_C$. As the successive edges of the stream are processed, H_C is updated if the input edge belongs to the component C. We now formally describe H_C and its properties.

Spanning subgraph H_C

As described earlier, at the beginning of a pass for every component C of G', $H_C = T_C$. Now, the role of H_C is to facilitate the maintenance of the invariant $\mathcal{I}_T$. In order to satisfy $\mathcal{I}_{T_1}$ and $\mathcal{I}_{T_2}$, we store in H_C all the edges in S_C that are incident on at least one vertex of

T'_C. Therefore, H_C is the spanning tree T_C along with every edge in S_C which has at least one endpoint in T'_C. Thus, H_C satisfies the following invariant throughout the algorithm.

Invariant $\mathcal{I}_H$:
H_C comprises of T_C and all edges from S_C that are incident on at least one vertex of T'_C.

We shall now describe a few properties of H_C and then in the following section show that maintaining $\mathcal{I}_H$ for H_C is indeed sufficient to maintain the invariant $\mathcal{I}_T$ as the stream is processed. The following properties of H_C are crucial to establish the correctness of our procedure to maintain T_C and H_C and establish a bound on total space required by H_C.

▶ **Lemma 8.** *T_C is a valid DFS tree of H_C.*

Proof. In order to prove this claim it is sufficient to prove that all the non-tree edges stored in H_C are back edges in T_C, i.e., the endpoints of every such edge share an ancestor-descendant relationship. Now, invariant $\mathcal{I}_{T_1}$ ensures that any edge in S_C having both endpoints in T'_C is a back edge. And invariant $\mathcal{I}_{T_2}$ ensures that any edge between a vertex in T'_C and $T_C \setminus T'_C$ is a back edge. Hence, all the non-tree edges incident on T'_C (and hence all non-tree edges in H_C) are back edges, proving our lemma. ◀

▶ **Lemma 9.** *The total number of edges in H_C, for all the components C of G', is $O(nk)$.*

Proof. The size of H_C can be analysed using invariant $\mathcal{I}_H$ as follows. The number of tree edges in T_C (and hence in H_C) is $O(|V_C|)$. The non-tree edges stored by H_C have at least one endpoint in T'_C. Using Lemma 8 we know that all these edges are back edges. To bound the number of such edges let us associate each non-tree edge to its lower endpoint. Hence each vertex will be associated to at most k non-tree edges to its k ancestors in T'_C (recall that T'_C is the top k levels of T_C). Thus, H_C stores $O(|V_C|)$ tree edges and $O(|V_C| \cdot k)$ non-tree edges, i.e., total $O(|V_C| \cdot k)$ edges. Since $\sum_{C \in G'} |V_C| \leq n$, the total number of edges in H_C is $O(nk)$. ◀

5.1 Processing of Edges

We now describe how T_C and H_C are maintained while processing the edges of the input stream such that $\mathcal{I}_T$ and $\mathcal{I}_H$ are satisfied. Since our algorithm maintains the invariants $\mathcal{I}_1$ and $\mathcal{I}_2$ (because of $\mathcal{I}_T$), we know that any edge whose both endpoints are not in some component C of G', is either a back edge or already a tree edge in T. Thus, we shall only discuss the processing of an edge (x, y) having both endpoints in C (now added to S_C), where $level(x) \leq level(y)$.

1. If $x \in T'_C$ then the edge is added to H_C to ensure $\mathcal{I}_H$. In addition, if (x, y) is a cross edge in T_C it violates either $\mathcal{I}_{T_1}$ (if $y \in T'_C$) or $\mathcal{I}_{T_2}$ (if $y \notin T'_C$). Thus, T_C is required to be restructured to ensure that $\mathcal{I}_T$ is satisfied.
2. If $x \notin T'_C$ and if x and y belong to different trees in $T_C \setminus T'_C$, then it violates $\mathcal{I}_{T_2}$. Again in such a case, T_C is required to be restructured to ensure that $\mathcal{I}_T$ is satisfied.

Note that after restructuring T_C we need to update H_C such that $\mathcal{I}_H$ is satisfied. Consequently any non-tree edge in H_C that was incident on a vertex in original T'_C, has to be removed from H_C if none of its endpoints are in T'_C after restructuring T_C, i.e., one or both of its endpoints have moved out of T'_C. But the problem arises if a vertex moves into T'_C during restructuring. There might have been edges incident on such a vertex in S_C and which were not stored in H_C. In this case we need these edges in H_C to satisfy $\mathcal{I}_H$, which is not possible without visiting S_C again. This problem can be avoided if our restructuring procedure ensures that no new vertex enters T'_C. This can be ensured if the restructuring procedure follows the property of *monotonic fall*, i.e., the level of a vertex is never decreased by the procedure. Let e be the new edge of component C in the input stream. We shall show that in order to preserve the invariants $\mathcal{I}_T$ and $\mathcal{I}_H$ it is sufficient that the restructuring procedure maintains the property of *monotonic fall* and ensures that the restructured T_C is a DFS tree of $H_C + e$.

▶ **Lemma 10.** *On insertion of an edge e, any restructuring procedure which updates T_C to be a valid DFS tree of $H_C + e$ ensuring* monotonic fall*, satisfies the invariants $\mathcal{I}_T$ and $\mathcal{I}_H$.*

Proof. The property of *monotonic fall* ensures that the vertex set of new T'_C is a subset of the vertex set of the previous T'_C. Using $\mathcal{I}_H$ we know that any edge of S_C which is not present in H_C must have both its endpoints outside T'_C. Hence, *monotonic fall* guarantees that $\mathcal{I}_H$ continues to hold with respect to the new T'_C for the edges in $S_C \setminus \{e\}$. Additionally, we save e in the new H_C if at least one of its endpoints belong to the new T'_C, ensuring that $\mathcal{I}_H$ holds for the entire S_C.

Since the restructuring procedure ensures that the updated T_C is a DFS tree of H_C, the invariant $\mathcal{I}_{T_1}$ trivially holds as a result of $\mathcal{I}_H$. In order to prove the invariant $\mathcal{I}_{T_2}$, consider any edge $e' \in S_C$ from a tree $\tau \in T_C \setminus T'_C$ to $T_C \setminus \tau$. Clearly, it will satisfy $\mathcal{I}_{T_2}$ if $e' \in H_C$, as T_C is a DFS tree of $H_C + e$. In case $e' \notin H_C$, it must be internal to some tree τ' in the original $T_C \setminus T'_C$ (using $\mathcal{I}_{T_2}$ in the original T_C). We shall now show that such an edge will remain internal to some tree in the updated $T_C \setminus T'_C$ as well, thereby not violating $\mathcal{I}_{T_2}$. Clearly the endpoints of e' cannot be in the updated T'_C due to the property of *monotonic fall*.

Assume that the endpoints of e' belong to different trees of updated $T_C \setminus T'_C$. Now, consider the edges $e_1, ..., e_t$ on the tree path in τ' connecting the endpoints of e'. Since the entire tree path is in τ', the endpoints of each e_i are not in original T'_C, ensuring that they are also not in the updated T'_C (by *monotonic fall*). Since the endpoints of e' (and hence the endpoints of the path $e_1, ..., e_t$) are in different trees in updated $T_C \setminus T'_C$, there must exist some e_i which also has endpoints belonging to different trees of updated $T_C \setminus T'_C$. This makes e_i a cross edge of the updated T_C. Since e_i is a tree edge of original T_C, it belongs to H_C and hence e_i being a cross edge implies that the updated T_C is not a DFS tree of $H_C + e$, which is a contradiction. Hence e' has both its endpoints in the same tree of the updated $T_c \setminus T'_C$, ensuring that $\mathcal{I}_{T_2}$ holds after the restructuring procedure. ◀

Hence, any procedure to restructure a DFS tree T_C of the subgraph H_C on insertion of a *cross edge* e, that upholds the property of *monotonic fall* and returns a new T_C which is a DFS tree of $H_C + e$, can be used as a *black box* in our algorithm. One such algorithm is the incremental DFS algorithm by Baswana and Khan [8], which precisely fulfils our requirement. They proved the total update time of the algorithm to be $O(n^2)$. They also showed that any algorithm maintaining incremental DFS abiding *monotonic fall* would require $\Omega(n^2)$ time even for sparse graphs, if it explicitly maintains the DFS tree. If the height of the DFS tree is known to be h, these bounds reduces to $O(nh + n_e)$ and $\Omega(nh + n_e)$ respectively, where n_e is the number of edges processed by the algorithm. Refer to the full paper for a brief description of the algorithm.

5.2 Algorithm

We now describe the details of our final algorithm which uses an incremental DFS algorithm [8] for restructuring the DFS tree when a cross edge is inserted. Similar to the algorithm in Section 4, for each component C of G', a rooted spanning tree T_C of the component is required as an input to the procedure having the root r_C.

Initially $T = \emptyset$ and $G' = G$ has a single component C, as G is connected (recall the assumption in Section 2). Hence for the first pass, we compute a spanning tree T_C of G using the Union-Find algorithm. Subsequently in each pass we directly get a spanning tree $T_{C'}$ for each component C' of the new G', which is the corresponding tree in $T_C \setminus T'_C$, where C is the component containing C' in the previous pass. Also, observe that the use of these trees as the new T_C ensures that the level of no vertex ever rises in the context of the entire tree T. This implies that the level of any vertex starting with the initial spanning tree T_G never rises, i.e., the entire algorithm satisfies the property of *monotonic fall*. We will use this fact crucially in the analysis of the time complexity.

As described earlier, we process the edges of the stream by updating the T_C and H_C maintaining $\mathcal{I}_T$ and $\mathcal{I}_H$ respectively. In case the edge is internal to some tree in $T_C \setminus T'_C$ (i.e., have both endpoint in the same tree in $T_C \setminus T'_C$), we simply ignore the edge. Otherwise, we add it to H_C to satisfy $\mathcal{I}_H$. Further, the incremental DFS algorithm [8] maintains T_C to be a DFS tree of H_C, which restructures T_C if the processed edge is added to H_C and is a cross edge in T_C. Now, in case T_C is updated we also update the subgraph H_C, by removing the extra non-tree edges having both endpoints in $T_C \setminus T'_C$. After the pass is completed, we attach T'_C (the top k levels of T_C) to T. Now, $\mathcal{I}_{T_2}$ ensures that each tree of $T_C \setminus T'_C$ forms the (rooted) spanning tree of the components of the new G', and hence can be used for the next pass. Refer to the full paper for the pseudocode of the algorithm.

5.3 Correctness and Analysis

The correctness of our algorithm follows from Lemma 10, which ensures that invariants $\mathcal{I}_H$ and $\mathcal{I}_T$ (and hence $\mathcal{I}_1$ and $\mathcal{I}_2$) are maintained as a result of using the incremental DFS algorithm which ensures *monotonic fall* of vertices. The total space used by our algorithm and the restructuring procedure is dominated by the cumulative size of H_C for all components C of G', which is $O(nk)$ using Lemma 9. Now, in every pass of the algorithm, a DFS tree for each component C of height k is attached to T. These trees collectively constitute the next k levels of the final DFS tree T. Therefore, the entire tree T is computed in $\lceil h/k \rceil$ passes.

Let us now analyse the time complexity of our algorithm. In the first pass $O(m\alpha(m,n))$ time is required to compute the spanning tree T_C using the Union-Find algorithm. Also, in each pass $O(m)$ time is required to process the input stream. Further, in order to update H_C we are required to delete edges having both endpoints out of T'_C. Hence, whenever a vertex falls below the k^{th} level, the edges incident on it are checked for deletion from H_C (if the other endpoint is also not in T'_C). Total time required for this is $O(\sum_{v \in V} deg(v)) = O(m)$ per pass. In the full paper we describe the details of an incremental DFS algorithm which maintains the DFS tree in total $O(nh + n_e)$ time, where $n_e = O(mh/k)$, for processing the entire input stream in each pass.

Finally, we need to efficiently answer the *query* whether an edge is internal to some tree in $T_C \setminus T'_C$. For this we maintain for each vertex x its ancestor at level k as $rep[x]$, i.e., $rep[x]$ is the root of the tree in $T_C \setminus T'_C$ that contains x. If $level(x) < k$, then $rep[x] = x$. For an edge (x, y) comparing the $rep[x]$ and $rep[y]$ efficiently answers the required query in $O(1)$ time. However, whenever T_C is updated we need to update $rep[v]$ for each vertex v in the modified part of T_C, requiring $O(1)$ time per vertex in the modified part of T_C. We shall bound the total work done to update $rep[x]$ of such a vertex x throughout the algorithm to $O(nh)$ as follows.

Consider the potential function $\Phi = \sum_{v \in V} level(v)$. Whenever some part of T_C is updated, each vertex x in the modified T_C necessarily incurs a fall in its level (due to *monotonic fall*). Thus, the cost of updating $rep[x]$ throughout the algorithm is proportional to the number of times x descends in the tree, hence increases the value of Φ by at least one unit. Hence, updating $rep[x]$ for all x in the modified part of T_C can be accounted by the corresponding increase in the value of Φ. Clearly, the maximum value of Φ is $O(nh)$, since the level of each vertex is always less than h, where h is the height of the computed DFS tree. Thus, the total work done to update $rep[x]$ for all $x \in V$ is $O(nh)$. This proves our main theorem described in Section 1.1 which is stated as follows.

▶ **Theorem 2.** *Given an undirected graph G, a DFS tree of G can be computed by a semi-streaming algorithm using $\lceil h/k \rceil$ passes using $O(nk)$ space requiring amortized $O(m + nk)$ time per pass for any integer $k \leq h$, where h is the height of the computed DFS tree.*

▶ Remark 11. Note that the time complexity of our algorithm is indeed tight for our framework. Since our algorithm requires $\lceil h/k \rceil$ passes and any restructuring procedure following *monotonic fall* requires $\Omega(nh + n_e)$ time, each pass would require $\Omega(m + nk)$ time.

6 Experimental Evaluation

Most streaming algorithms deal with only $O(n)$ space, for which our advanced algorithms improve over the simple algorithms theoretically by just constant factors. However, their empirical performance demonstrates their significance in the real world applications. The evaluation of our algorithms on random and real graphs shows that in practice these algorithms require merely a *few* passes even when allowed to store just $5n$ edges. The results of our analysis can be summarized as follows (for details refer to the full paper).

The two advanced algorithms *kPath* (algorithmin Section 4) and *kLev* (algorithm in Section 5 with an additional heuristic) perform much better than the rest even when $O(n)$ space is allowed. For both random and real graphs, *kPath* performs slightly worse as the density of the graph increases. On the other hand *kLev* performs slightly better only in random graphs with the increasing density. The effect of the space parameter is very large on *kPath* from $k = 1$ to small constants, requiring very *few* passes even for $k = 5$ and $k = 10$. However, *kLev* seems to work very well even for $k = 1$ and has a negligible effect of increasing the value of k. Overall, the results suggest using *kPath* if nk space is allowed for k being a small constant such as 5 or 10. However, if the space restrictions are extremely tight it is better to use *kLev*.

7 Conclusion

We presented the first $o(n)$ pass semi-streaming algorithm for computing a DFS tree for an undirected graph, breaking the long standing presumed barrier of n passes. In our streaming model we assume that $O(nk)$ local space is available for computation, where k is any natural number. Our algorithm computes a DFS tree in $\lceil n/k \rceil$ passes. We improve our algorithm to require only $\lceil h/k \rceil$ passes without any additional space requirement, where h is the height of the final tree. This improvement becomes significant for graphs having shallow DFS trees. Moreover, our algorithm is described as a *framework* using a restructuring algorithm as a black box. This allows more flexibility to extend our algorithm for solving other problems requiring a computation of DFS tree in the streaming environment.

Recently, in a major breakthrough Elkin [12] presented the first $o(n)$ pass algorithm for computing Shortest Paths Tree from a single source. Using $O(nk)$ local space, it computes the shortest path tree from a given source in $O(n/k)$ passes for unweighted graphs and in $O(n \log n/k)$ passes for weighted graphs.

Despite the fact that these breakthroughs provide only minor improvements (typically $poly \log n$ factors), they are significant steps to pave a path in better understanding of such fundamental problems in the streaming environment. These simple improvements come after decades of the emergence of streaming algorithms for graph problems, where such problems were considered implicitly hard in the semi-streaming environment. We thus believe that our result is a significant improvement over the known algorithm for computing a DFS tree in the streaming environment, and it can be a useful step in more involved algorithms that require the computation of a DFS tree.

Moreover, the experimental evaluation of our algorithms revealed exceptional performance of the advanced algorithms *kPath* and *kLev* (greatly affected by the *additional heuristic*). Thus, it would be interesting to further study these algorithms theoretically which seem to work extremely well in practice.

References

1 Kook Jin Ahn and Sudipto Guha. Linear programming in the semi-streaming model with application to the maximum matching problem. *Inf. Comput.*, 222:59–79, 2013.

2 Noga Alon, Yossi Matias, and Mario Szegedy. The Space Complexity of Approximating the Frequency Moments. *Journal of Computer and System Sciences*, 58(1):137–147, 1999.

3 Giorgio Ausiello, Donatella Firmani, and Luigi Laura. Datastream computation of graph biconnectivity: Articulation Points, Bridges, and Biconnected Components. In *Theoretical Computer Science, 11th Italian Conference, ICTCS 2009, Cremona, Italy, September 28-30, 2009, Proceedings.*, pages 26–29, 2009.

4 Giorgio Ausiello, Donatella Firmani, and Luigi Laura. Real-time monitoring of undirected networks: Articulation points, bridges, and connected and biconnected components. *Networks*, 59(3):275–288, 2012. doi:10.1002/net.21450.

5 Ziv Bar-Yossef, Ravi Kumar, and D. Sivakumar. Reductions in Streaming Algorithms, with an Application to Counting Triangles in Graphs. In *Proceedings of the Thirteenth Annual ACM-SIAM Symposium on Discrete Algorithms*, SODA '02, pages 623–632, 2002.

6 Surender Baswana. Streaming algorithm for graph spanners - single pass and constant processing time per edge. *Inf. Process. Lett.*, 106(3):110–114, 2008.

7 Surender Baswana, Shreejit Ray Chaudhury, Keerti Choudhary, and Shahbaz Khan. Dynamic DFS in Undirected Graphs: breaking the O(m) barrier. In *ACM-SIAM Symposium on Discrete Algorithms, SODA*, pages 730–739, 2016.

8 Surender Baswana and Shahbaz Khan. Incremental Algorithm for Maintaining a DFS Tree for Undirected Graphs. *Algorithmica*, 79(2):466–483, 2017.

9 Glencora Borradaile, Claire Mathieu, and Theresa Migler. Lower bounds for testing digraph connectivity with one-pass streaming algorithms. *CoRR*, abs/1404.1323, 2014.

10 Adam L. Buchsbaum, Raffaele Giancarlo, and Jeffery Westbrook. On finding common neighborhoods in massive graphs. *Theor. Comput. Sci.*, 299(1-3):707–718, 2003.

11 Michael Elkin. Streaming and fully dynamic centralized algorithms for constructing and maintaining sparse spanners. *ACM Trans. Algorithms*, 7(2):20:1–20:17, 2011.

12 Michael Elkin. Distributed exact shortest paths in sublinear time. In *Proceedings of the 49th Annual ACM SIGACT Symposium on Theory of Computing, STOC 2017, Montreal, QC, Canada, June 19-23, 2017*, pages 757–770, 2017.

13 Shimon Even and Robert Endre Tarjan. Network Flow and Testing Graph Connectivity. *SIAM J. Comput.*, 4(4):507–518, 1975.

14 Martin Farach-Colton, Tsan-sheng Hsu, Meng Li, and Meng-Tsung Tsai. Finding Articulation Points of Large Graphs in Linear Time. In *Algorithms and Data Structures, WADS*, pages 363–372, Cham, 2015. Springer International Publishing.

15 Joan Feigenbaum, Sampath Kannan, Andrew McGregor, Siddharth Suri, and Jian Zhang. Graph Distances in the Streaming Model: The Value of Space. In *Proceedings of the Sixteenth Annual ACM-SIAM Symposium on Discrete Algorithms*, SODA '05, pages 745–754, 2005.

16 Joan Feigenbaum, Sampath Kannan, Andrew McGregor, Siddharth Suri, and Jian Zhang. On Graph Problems in a Semi-streaming Model. *Theor. Comput. Sci.*, 348(2):207–216, December 2005.

17 Joan Feigenbaum, Sampath Kannan, Martin J. Strauss, and Mahesh Viswanathan. An Approximate L1-Difference Algorithm for Massive Data Streams. *SIAM J. Comput.*, 32(1):131–151, January 2003.

18 Philippe Flajolet and G. Nigel Martin. Probabilistic counting algorithms for data base applications. *Journal of Computer and System Sciences*, 31(2):182–209, 1985.

19 Sudipto Guha, Nick Koudas, and Kyuseok Shim. Data-streams and Histograms. In *Proceedings of the Thirty-third Annual ACM Symposium on Theory of Computing*, STOC '01, pages 471–475, 2001.

20 Venkatesan Guruswami and Krzysztof Onak. Superlinear Lower Bounds for Multipass Graph Processing. *Algorithmica*, 76(3):654–683, November 2016.

21 Venkatesan Guruswami and Krzysztof Onak. Superlinear Lower Bounds for Multipass Graph Processing. *Algorithmica*, 76(3):654–683, 2016.

22 Monika Rauch Henzinger, Prabhakar Raghavan, and Sridhar Rajagopalan. Computing on data streams. In *External Memory Algorithms, Proceedings of a DIMACS Workshop, New Brunswick, New Jersey, USA, May 20-22, 1998*, pages 107–118, 1998.

23 John E. Hopcroft and Richard M. Karp. An $n^{5/2}$ Algorithm for Maximum Matchings in Bipartite Graphs. *SIAM J. Comput.*, 2(4):225–231, 1973.

24 John E. Hopcroft and Robert Endre Tarjan. Efficient Planarity Testing. *J. ACM*, 21(4):549–568, 1974.

25 Piotr Indyk. Stable Distributions, Pseudorandom Generators, Embeddings, and Data Stream Computation. *J. ACM*, 53(3):307–323, May 2006.

26 Sagar Kale and Sumedh Tirodkar. Maximum Matching in Two, Three, and a Few More Passes Over Graph Streams. In *Approximation, Randomization, and Combinatorial Optimization. Algorithms and Techniques, APPROX/RANDOM 2017, August 16-18, 2017, Berkeley, CA, USA*, pages 15:1–15:21, 2017.

27 Michael Kapralov. Better bounds for matchings in the streaming model. In *Proceedings of the Twenty-Fourth Annual ACM-SIAM Symposium on Discrete Algorithms, SODA 2013, New Orleans, Louisiana, USA, January 6-8, 2013*, pages 1679–1697, 2013.

28 Shahbaz Khan. Near Optimal Parallel Algorithms for Dynamic DFS in Undirected Graphs. In *Proceedings of the 29th ACM Symposium on Parallelism in Algorithms and Architectures, SPAA 2017, Washington DC, USA, July 24-26, 2017*, pages 283–292, 2017.

29 Lasse Kliemann. Engineering a Bipartite Matching Algorithm in the Semi-Streaming Model. In *Algorithm Engineering - Selected Results and Surveys*, pages 352–378. Springer International Publishing, 2016.

30 Andrew McGregor. Finding Graph Matchings in Data Streams. In *Approximation, Randomization and Combinatorial Optimization, Algorithms and Techniques, 8th International Workshop on Approximation Algorithms for Combinatorial Optimization Problems, APPROX 2005 and 9th InternationalWorkshop on Randomization and Computation, RANDOM 2005, Berkeley, CA, USA, August 22-24, 2005, Proceedings*, pages 170–181, 2005.

31 Andrew McGregor. Graph Stream Algorithms: A Survey. *SIGMOD Rec.*, 43(1):9–20, May 2014.

32 Shan Muthukrishnan. Data Streams: Algorithms and Applications. *Foundations and Trends in Theoretical Computer Science*, 1(2):117–236, 2005.

33 Thomas C. O'Connell. A Survey of Graph Algorithms Under Extended Streaming Models of Computation. In *Fundamental Problems in Computing: Essays in Honor of Professor Daniel J. Rosenkrantz*, pages 455–476, 2009.

34 Jan Matthias Ruhl. Efficient Algorithms for New Computational Models. *PhD Thesis*, Department of Computer Science, MIT, Cambridge, MA, 2003.

35 Robert Endre Tarjan. Depth-First Search and Linear Graph Algorithms. *SIAM J. Comput.*, 1(2):146–160, 1972.

36 Robert Endre Tarjan. Finding Dominators in Directed Graphs. *SIAM J. Comput.*, 3(1):62–89, 1974.

37 Robert Endre Tarjan. Efficiency of a Good But Not Linear Set Union Algorithm. *J. ACM*, 22(2):215–225, April 1975.

38 Robert Endre Tarjan. Edge-Disjoint Spanning Trees and Depth-First Search. *Acta Inf.*, 6:171–185, 1976.

39 Robert Endre Tarjan and Jan van Leeuwen. Worst-case Analysis of Set Union Algorithms. *J. ACM*, 31(2):245–281, 1984.

40 Jeffery Westbrook and Robert Endre Tarjan. Maintaining Bridge-Connected and Biconnected Components On-Line. *Algorithmica*, 7(5&6):433–464, 1992.

41 Jian Zhang. A Survey on Streaming Algorithms for Massive Graphs. In *Managing and Mining Graph Data*, pages 393–420. Springer US, 2010.