Near Optimal Parallel Algorithms for Dynamic DFS in Undirected Graphs*

Shahbaz Khan
Department of CSE, IIT Kanpur
Kanpur, UP, India 208016
shahbazk@cse.iitk.ac.in

ABSTRACT

Depth first search (DFS) tree is a fundamental data structure for solving various graph problems. The classical algorithm [37] for building a DFS tree requires $O(m+n)$ time for a given undirected graph G having n vertices and m edges. Recently, Baswana et al. [5] presented a simple algorithm for updating the DFS tree of an undirected graph after an edge/vertex update in $\tilde{O}(n)$ [1] time. However, their algorithm is strictly sequential. We present an algorithm achieving similar bounds that can be easily adopted to the parallel environment.

In the parallel environment, a DFS tree can be computed from scratch using $O(m)$ processors in expected $\tilde{O}(1)$ time [2] on an EREW PRAM, whereas the best deterministic algorithm takes $\tilde{O}(\sqrt{n})$ time [2, 17] on a CRCW PRAM. Our algorithm can be used to develop optimal time (upto *poly* $\log n$ factors) deterministic parallel algorithms for maintaining fully dynamic DFS and fault tolerant DFS of an undirected graph.

(1) *Parallel Fully Dynamic DFS*:
Given any arbitrary online sequence of vertex or edge updates, we can maintain a DFS tree of an undirected graph in $\tilde{O}(1)$ time per update using m processors on an EREW PRAM.
(2) *Parallel Fault tolerant DFS*:
An undirected graph can be preprocessed to build a data structure of size $O(m)$, such that for any set of k updates (where k is constant) in the graph, a DFS tree of the updated graph can be computed in $\tilde{O}(1)$ time using n processors on an EREW PRAM. For constant k, this is also work optimal (upto *poly* $\log n$ factors).

Moreover, our fully dynamic DFS algorithm provides, in a seamless manner, nearly optimal (upto *poly* $\log n$ factors) algorithms for maintaining a DFS tree in the semi-streaming environment and a restricted distributed model. These are the first parallel, semi-streaming and distributed algorithms for maintaining a DFS tree in the dynamic setting.

*The full version of the paper is available in [23]
[1] $\tilde{O}()$ hides the poly-logarithmic factors.

SPAA '17, July 24-26, 2017, Washington DC, USA
© 2017 Copyright held by the owner/author(s). Publication rights licensed to Association for Computing Machinery.
ACM ISBN 978-1-4503-4593-4/17/07…$15.00
http://dx.doi.org/10.1145/3087556.3087576

CCS CONCEPTS

• **Theory of computation → Dynamic graph algorithms**; **Parallel algorithms**; Streaming, sublinear and near linear time algorithms; Distributed algorithms; • **Mathematics of computing →** *Graph algorithms*;

KEYWORDS

parallel, dynamic, DFS, graph, algorithm, streaming, distributed

1 INTRODUCTION

Depth First Search (DFS) is a well known graph traversal technique. Right from the seminal work of Tarjan [37], DFS traversal has played a central role in the design of efficient algorithms for many fundamental graph problems, namely, strongly connected components, topological sorting [39], dominators in directed graph [38], edge and vertex connectivity [11] etc.

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 called a DFS tree rooted at r 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. Otherwise, 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. Thus, many DFS trees are possible for any given graph from a given root r. However, if the traversal is performed strictly according to the order of edges in the adjacency lists of the graph, the resulting DFS tree will be unique. Ordered DFS tree problem is to compute the order in which the vertices are visited by this unique DFS traversal.

An algorithmic graph problem is modeled in a dynamic environment as follows. There is an online sequence of updates on the graph, and the objective is to update the solution of the problem efficiently after each update. In particular, the time taken to update the solution has to be much smaller than that of the best static algorithm for the problem. A dynamic graph algorithm is said to be *fully dynamic* if it handles both insertion and deletion updates, otherwise it is called *partially dynamic*. Another, and more restricted, variant of a dynamic environment is the fault tolerant environment. Here the aim is to build a compact data structure, for a given problem, that is resilient to failures of vertices/edges and can efficiently report the solution after a given set of failures.

Recently, Baswana et al. [5, 6] presented a fully dynamic algorithm for maintaining a DFS tree of an undirected graph in $\tilde{O}(\sqrt{mn})$ time per update. They also presented an algorithm for updating the DFS tree after a single update in $\tilde{O}(n)$ time. Prior to this work, only partially dynamic algorithms were known for DFS trees [7, 8, 14].

Now, major applications of dynamic graphs in the real world involve a huge amount of data, which makes recomputing the solution after every update infeasible. Due to this large size of data, it also becomes impractical for solving such problems on a single sequential machine because of both memory and computation costs involved. Thus, it becomes more significant to explore these dynamic graph problems on a computation model that efficiently handles large storage and computations. In the past three decades a lot of work has been done to address dynamic graph problems in parallel [9, 13, 33, 35], semi-streaming [3, 18, 21, 28], and distributed (also called dynamic networks) [4, 22, 24, 36] environments.

In this paper, we address the problem of maintaining dynamic DFS tree efficiently in the parallel environment and demonstrate its applications in semi-streaming and distributed environments.

1.1 Existing results

In spite of the simplicity of a DFS tree, designing efficient parallel, distributed or streaming algorithms for a DFS tree has turned out to be quite challenging. Reif [31] showed that the ordered DFS tree problem is P-Complete. For many years, this result seemed to imply that general DFS tree problem, i.e., computing any DFS tree of the graph, is also inherently sequential. However, Aggarwal et al. [1, 2] proved that the general DFS tree problem is in RNC[2] by designing a randomized EREW PRAM[3] algorithm that takes $\tilde{O}(1)$ time. But the fastest deterministic algorithm for computing general DFS tree in parallel still takes $\tilde{O}(\sqrt{n})$ time [2, 17] in CRCW PRAM[4], even for undirected graphs. Moreover, general DFS tree problem has been shown to be in NC for some special graphs including DAGs [16] and planar graphs [19] (see [15] for a survey). Whether general DFS tree problem is in NC is still a long standing open problem.

In the semi-streaming environment, the input graph is accessed in form of a stream of graph edges, where an algorithm can perform multiple passes on this stream but is allowed to use only $O(n)$ local space. The DFS tree can be trivially computed using $O(n)$ passes, where each pass adds one vertex to the DFS tree. However, computing the DFS tree in $\tilde{O}(1)$ passes is considered hard [12]. To the best of our knowledge, it is an open problem to compute a DFS tree using even $o(n)$ passes in any relaxed streaming environment [29, 32].

Computing a DFS tree in a distributed setting was widely studied in 1980's and 1990's. A DFS tree can be computed in $O(n)$ rounds, with different trade offs between number of messages passed, and size of each message. A DFS tree can be built using $O(n)$ messages if we allow messages of size $O(n)$ [25, 27, 34], and using $O(m)$ messages if we limit the message size to be $\tilde{O}(1)$ [10, 26, 40].

Thus, to maintain a DFS tree in the dynamic setting, each update requires $\tilde{O}(\sqrt{n})$ time on a CRCW PRAM in deterministic parallel setting, $O(n)$ passes in semi-streaming setting and $O(n)$ rounds in distributed setting, which is very inefficient. Hence, exploring the dynamic DFS in parallel, semi-streaming and distributed models seems to be a long neglected problem of practical significance.

[2]NC is the class of problems solvable using $O(n^{c_1})$ processors in parallel $O(\log^{c_2} n)$ time, for any constants c_1, c_2. The class RNC extends NC to allow access to randomness.
[3]Exclusive Read Exclusive Write (EREW) restricts any two processors to simultaneously read or write the same memory cell. Concurrent Read Concurrent Write (CRCW) does not have this restriction.
[4] It essentially shows DFS to be NC equivalent of minimum-weight perfect matching, which is in RNC whereas its best deterministic algorithm requires $\tilde{O}(\sqrt{n})$ time.

We present optimal algorithms (up to *poly* $\log n$ factors) for maintaining a fully dynamic DFS tree for an undirected graph under both edge and vertex updates on these models.

1.2 Our Results

We consider an extended notion of updates wherein an update could be either insertion/deletion of a vertex or insertion/deletion of an edge. Furthermore, an inserted vertex can be added with any set of incident edges to the graph. In the parallel setting, our main result can be succinctly described as follows.

Theorem 1.1. *Given an undirected graph and its DFS tree, it can be preprocessed to build a data structure of size $O(m)$ in $O(\log n)$ time using m processors on an EREW PRAM, such that for any update in the graph, a DFS tree of the updated graph can be computed in $O(\log^3 n)$ time using n processors on an EREW PRAM.*

With this result at the core, we easily obtain the following results.

(1) *Parallel Fully Dynamic DFS*:
Given an arbitrary online sequence of updates, we can maintain a DFS tree of an undirected graph in $O(\log^3 n)$ time per update using m processors on an EREW PRAM.
(2) *Parallel Fault tolerant DFS*:
An undirected graph can be preprocessed to build a data structure of size $O(m)$, such that for any set of $k(\leq \log n)$ updates in the graph, a DFS tree of the updated graph can be computed in $O(k \log^{2k+1} n)$ time using n processors on an EREW PRAM.

Our fully dynamic algorithm and fault tolerant algorithm (for constant k), clearly take optimal time (up to *poly* $\log n$ factors) for maintaining a DFS tree. Our fault tolerant algorithm (for constant k) is also work optimal (upto *poly* $\log n$ factors) since a single update can lead to $\Theta(n)$ changes in the DFS tree. Moreover, our result also establishes that maintaining fully dynamic DFS for an undirected graph is in NC (which is still an open problem for static DFS).

1.3 Applications of Parallel Fully Dynamic DFS

Our parallel fully dynamic DFS algorithm can be seamlessly adapted to the semi-streaming and distributed environments as follows.

(1) *Semi-streaming Fully Dynamic DFS*:
Given any arbitrary online sequence of vertex or edge updates, we can maintain a DFS tree of an undirected graph using $O(\log^2 n)$ passes over the input graph per update by a semi-streaming algorithm using $O(n)$ space.
(2) *Distributed Fully Dynamic DFS*:
Given any arbitrary online sequence of vertex or edge updates, we can maintain a DFS tree of an undirected graph in $O(D \log^2 n)$ rounds per update in the synchronous $\mathcal{CONGEST}(n/D)$ model using $O(nD \log^2 n + m)$ messages of size $O(n/D)$ requiring $O(n)$ space on each processor, where D is diameter of the graph.

Our semi-streaming algorithm clearly takes optimal number of passes (up to *poly* $\log n$ factors) for maintaining a DFS tree. Our distributed algorithm that works in a restricted $CONGEST(B)$ [5]

[5]$CONGEST(B)$ model is the standard $\mathcal{CONGEST}$ model [30] where message size is relaxed to B words.

model, also arguably requires optimal rounds (up to *poly* $\log n$ factors) because it requires $\Omega(D)$ rounds to propagate the information of the update throughout the graph. Since almost the whole DFS tree may need to be updated due to a single update in the graph, every algorithm for maintaining a DFS tree in the distributed setting will require $\Omega(D)$ rounds [6]. This essentially improves the state of the art for the classes of graphs with $o(n)$ diameter.

1.4 Overview

We now describe a brief overview of our result. Baswana et al. [5] proved that updating a DFS tree after any update in the graph is equivalent to *rerooting* disjoint subtrees of the DFS tree. They also presented an algorithm to reroot a DFS tree T (or its subtree), originally rooted at r to a new root r', in $\tilde{O}(n)$ time. It starts the traversal from r' traversing the path connecting r' to r in T. Now, the subtrees hanging from this path are essentially the components of the *unvisited graph* (the subgraph induced by the unvisited vertices of the graph) due to the absence of *cross edges*. In the updated DFS tree, every such subtree, say τ, shall hang from an edge emanating from τ to the path from r' to r. Let this edge be (x, y), where $x \in \tau$. Thus, we need to recursively reroot τ to the new root x and hang it from (x, y) in the updated DFS tree. Note that this rerooting can be independently performed for different subtrees hanging from tree path from r' to r.

At the core of their result, they use a property of the DFS tree, that they called *components* property, to find the edge (x, y) efficiently, using a data structure $\mathcal{D}_0$. However, as evident from the discussion above, their rerooting procedure can be strictly sequential in the worst case. This is because the size of a subtree τ to be rerooted can be almost equal to that of the original tree T. As a result, $O(n)$ sequential reroots may be required in the worst case. Our main contribution is an algorithm that performs this rerooting efficiently in parallel.

Our algorithm ensures that rerooting is completed in $\tilde{O}(1)$ steps as follows. At any point of time, we ensure that every component c of the *unvisited graph* is either of type $C1$, having a single subtree of T, or of type $C2$, having a path p_c and a set of subtrees of T having edges to p_c. Note that in [5] every component of the unvisited graph is of type $C1$. We define three types of traversals, namely, *path halving* (also used by [5]), *disintegrating traversal* and *disconnecting traversal*. We prove that using a combination of $O(1)$ such traversals, for every component c of the unvisited graph, either the length of p_c is halved or the size of largest subtree in c is halved. Moreover, these traversals can be performed in $O(\log n)$ time on $|c|$ processors using the *components* property and a data structure $\mathcal{D}$ (answering similar queries as $\mathcal{D}_0$). However, since our algorithm ensures that each vertex is queried by $\mathcal{D}$ only $\tilde{O}(1)$ times (unlike [5]), our data structure $\mathcal{D}$ is much simpler than $\mathcal{D}_0$.

Furthermore, both our algorithm and the algorithm by [5] use the non-tree edges of the graph only to answer queries on data structure $\mathcal{D}$ (or $\mathcal{D}_0$). The remaining operations (except for queries on $\mathcal{D}$) required by our algorithm can be performed using only edges of T in $O(n)$ space. As a result, our algorithm being efficient in parallel setting (unlike [5]), can also be adapted to the semi-streaming and distributed model as follows. In the semi-streaming model, the passes over the input graph are used only to answer the queries on $\mathcal{D}$, where the parallel queries on $\mathcal{D}$ made by our algorithm can be answered simultaneously using a single pass. Our distributed algorithm only needs to store the current DFS tree at each node and the adjacency list of the corresponding vertex abiding the restriction of $O(n)$ space at each node. Again, the distributed computation is only used to answer queries on $\mathcal{D}$.

[6]For an algorithm maintaining the whole DFS tree at each node, even our message size is optimal. In the worst case, an update of size $O(n)$ (vertex insertion with arbitrary set of edges) have to be propagated throughout the network. In $O(D)$ rounds, it can only be propagated using messages of size $\Omega(n/D)$ (see full paper [23] for details).

2 PRELIMINARIES

Let $G = (V, E)$ be any given undirected graph on n vertices and m edges. The following notations will be used throughout the paper.

- $par(w)$: Parent of w in T.
- $T(x)$: The subtree of T rooted at vertex x.
- $path(x, y)$: Path from vertex x to vertex y in T.
- $LCA(x, y)$: Lowest common ancestor of x and y in T.
- $root(T')$: Root of a subtree T' of T, i.e., $root\big(T(x)\big) = x$.
- T^* : The updated DFS tree computed by our algorithm.

A subtree T' is said to be *hanging* from a path p if the $root(T')$ is a child of some vertex on the path p and does not belong to the path p. Unless stated otherwise, a component refers to a connected component of the unvisited graph. 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.

For our distributed algorithm, we use synchronous $CONGEST(B)$ model [30]. For the dynamic setting, Henzinger et al. [20] presented a model that has a *preprocessing* stage followed by an alternating sequence of non-overlapping stages for *update* and *recovery*. We use this model with an additional space restriction of $O(n)$ size at each node. Without this restriction, the whole graph can be stored at each node, where an algorithm can trivially propagate the update to each node and the updated solution can be computed locally. Also, we allow the deletion updates to be *abrupt*, i.e., the deleted link/node becomes unavailable for use instantly after the update.

In order to handle disconnected graphs, we add a dummy vertex r with edges to all the vertices. Our algorithm maintains a DFS tree rooted at r in this augmented graph, where each child subtree of r is a DFS tree of a connected component of the original graph.

We shall now define some queries that are performed by our algorithm on the data structure $\mathcal{D}$ (similar queries on $\mathcal{D}_0$ also used in [5]). Let $v, w, x, y \in V$, where $path(x, y)$ and $path(v, w)$ (if required) are ancestor-descendant paths in T. Also, no vertex in $path(v, w)$ is a descendant of any vertex in $path(x, y)$. We define the following queries.

(1) $Query\big(w, path(x, y)\big)$: among all the edges from w that are incident on $path(x, y)$ in G, return an edge that is incident nearest to x on $path(x, y)$.
(2) $Query\big(T(w), path(x, y)\big)$: among all the edges from $T(w)$ that are incident on $path(x, y)$ in G, return an edge that is incident nearest to x on $path(x, y)$.
(3) $Query\big(path(v, w), path(x, y)\big)$: among all the edges from $path(v, w)$ that are incident on $path(x, y)$ in G, return an edge that is incident nearest to x on $path(x, y)$.

Let the *descendant* vertices of the three queries described above be w, $T(w)$ and $path(v, w)$ respectively. A set of queries on the data structure $\mathcal{D}$ are called *independent* if the *descendant* vertices of these queries are disjoint.

Baswana et al. [5] described the *components* property as follows.

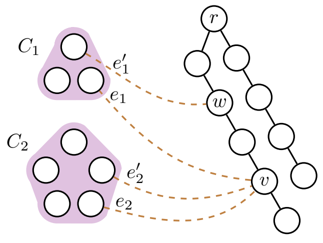

Figure 1: Edges e_1' and e_2' can be ignored during the DFS traversal (reproduced from [6]).

Lemma 2.1 (Components Property [5]). *Let T^* be the partially built DFS tree and v be the vertex currently being visited. Let $C_1, .., C_k$ be the connected components of the subgraph induced by the unvisited vertices. For any two edges e_i and e_i' from C_i that are incident respectively on v and some ancestor (not necessarily proper) w of v in T^*, 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). The edge e_i can be found by querying the data structure $\mathcal{D}$ (or $\mathcal{D}_0$ in [5]). The DFS tree is then updated after any update in the graph by reducing it to *rerooting* disjoint subtrees of the DFS tree using the components property. Rerooting a subtree $T(v)$ at a new root $r' \in T(v)$ involves restructuring the tree $T(v)$ to be now rooted at r', such that the new tree is also a DFS tree of the subgraph induced by $T(v)$. This reduction will henceforth be referred as the *reduction* algorithm and can be succinctly described as follow (see Appendix A for details).

Theorem 2.2. *Given an undirected graph G and its DFS tree T, any graph update can be reduced to independently rerooting disjoint subtrees of T by performing $O(1)$ sets of independent queries on the data structure $\mathcal{D}$ and $O(1)$ sets of LCA queries on T, where each set has at most n queries.*

3 REROOTING A DFS TREE

We now describe the algorithm to reroot a subtree $T(r_0)$ of the DFS tree T, from its original root r_0 to the new root r^*. Also, let the data structure $\mathcal{D}$ be built on T (see Section 2). We maintain the following invariant: at any moment of the algorithm, every component c of the unvisited graph can be of the following two types:

C1: Consists of a single subtree τ_c of the DFS tree T.

C2: Consists of a single *ancestor-descendant* path p_c and a set $\mathcal{T}_c$ of subtrees of the DFS tree T having at least one edge to p_c. Note that for any $\tau_1, \tau_2 \in \mathcal{T}_c$, there is no edge between τ_1 and τ_2 since T is a DFS tree.

Moreover, for every component c we also have a vertex $r_c \in c$ from which the DFS tree of the component c would be rooted in the final DFS tree T^*.

The algorithm is divided into $\log n$ *phases*, where each phase is further divided into $\log n$ *stages*. At the end of phase $\mathcal{P}_i$, every subtree of any component c (τ_c or subtrees in $\mathcal{T}_c$) has at most $n/2^i$ vertices. During phase $\mathcal{P}_i$, every component has at least one *heavy* subtree (having $> n/2^i$ vertices). If no such tree exists, we move the component to the next phase. We denote the set of these heavy subtrees by $\mathbb{T}_c$. For notational convenience, we refer to the heaviest subtree of every component c as τ_c, even for components of type $C2$. Hence, for any component of type $C1$ or $C2$, we have $\tau_c \in \mathbb{T}_c$. Clearly the algorithm ends after $\log n$ phases as every component of the unvisited graph would be empty.

At the end of stage $\mathcal{S}_j$ of a phase, the length of p_c in each component c is at most $n/2^j$. If $|p_c| \leq n/2^j$, we move the component c to the next stage. Further, for any component c of type $C1$, the value of $|p_c|$ is zero, so we move such components to the last stage of the phase, i.e., $\mathcal{S}_{\log n}$. Clearly at the end of $\log n$ stages, each component would be of type $C1$.

In the beginning of the algorithm, we have the component induced by $T(r_0)$ of type $C1$ where $r_c = r^*$. Note that during each stage, different connected components of the unvisited graph can be processed independent of each other in parallel.

Algorithm

We now describe how a component c in phase $\mathcal{P}_i$ and stage $\mathcal{S}_j$ is traversed by our algorithm. The aim is to build a partial DFS tree for the component c rooted at r_c, that can be attached to the partially built DFS tree T^* of the updated graph. Note that this has to be performed in such a manner that every component of the unvisited part of c is of type $C1$ or $C2$ only.

Now, in order to move to the next phase, we need to ensure that for every component c' of the unvisited part of c, $|\tau_{c'}| \leq n/2^i$. As described above, after $\log n$ stages every component c' is of type $C1$. Thus, we perform a *disintegrating traversal* of τ_c which ensures that every component of the unvisited part of c can be moved to the next phase.

During $\mathcal{S}_j$, in order to move to the next stage, we need to ensure that for every component c' of the unvisited part of c, either $|p_{c'}| \leq n/2^j$ (moving it to next stage) or $|\tau_{c'}| \leq n/2^i$ (moving it to next phase). The component is processed based on the location of r_c in c as follows. If $r_c \in p_c$, we perform *path halving* which ensures the components move to the next stage. If $r_c \in \tau \notin \mathbb{T}_c$, we perform a *disconnecting traversal* of τ followed by *path halving* of p_c, such that the unvisited components of τ are no longer connected to residual part of p_c, moving them to the next phase. The remaining components of c moves to the next stage due to path halving.

We shall refer to disintegrating traversal, path halving and disconnecting traversal as the *simpler* traversals. The difficult case is when $r_c \in \tau \in \mathbb{T}_c$. Here, some trivial cases can be directly processed by the *simpler* traversals mentioned above. For the remaining cases we perform *heavy subtree traversal* of τ which shall ensure that the unvisited part of c reduces to those requiring *simpler* traversals. Refer to the full paper [23] for pseudocodes of the algorithm.

We now describe the different types of traversals in detail. For any component c, we refer to the smallest subtree of $\tau \in \mathbb{T}_c$ that has more than $n/2^i$ vertices as $T(v_H)$. Since $n/2^{i-1} \geq |\tau| > n/2^i$, v_H is unique. Also, let $r' = root(\tau)$ (if $r_c \in \tau$) and $v_l = LCA(r_c, v_H)$.

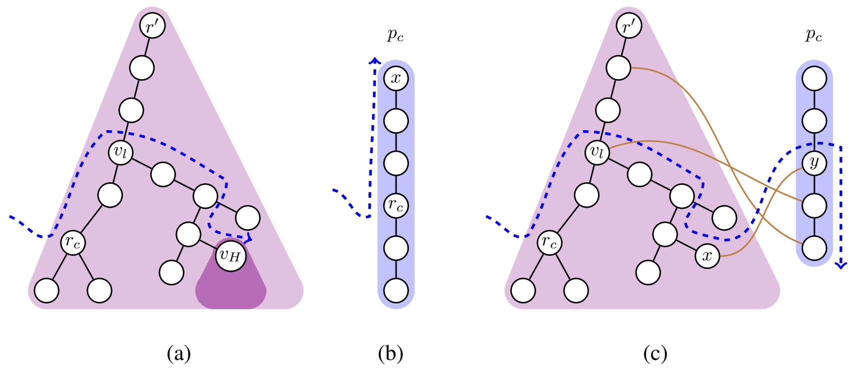

Figure 2: The three *simpler* traversals (shown using blue dotted lines), (a) Disintegrating traversal, (b) Path Halving, and (c) Disconnecting traversal.

3.1 Disintegrating Traversal

Consider a component c of type $C1$ with new root $r_c \in \tau_c$ in phase $\mathcal{P}_i$ ($n/2^i < |\tau_c| \leq n/2^{i-1}$). We first find the vertex v_H. We then traverse along the tree path $path(r_c, v_H)$, adding it to T^* (see Figure 2 (a)). Now, the unvisited part of c consists of $path(par(v_l), r')$ (say p) and the subtrees hanging from $path(r_c, r')$ and $path(v_l, v_H)$. Notice that p is an ancestor-descendant path of T and each subtree has at most $n/2^i$ vertices. Each subtree not having an edge to p corresponds to a separate component of type $C1$. The path p and the remaining subtrees (having an edge to p) form a component of type $C2$. For each component c^*, we also need to find the new root r_{c^*} for the updated DFS tree of the component. Using the components property, we know r_{c^*} has the lowest edge from c^* on the path p^*, where p^* is the newly attached path to T^* described above. Both these queries (finding an edge to p and the lowest edge on p^*) can be answered by our data structure $\mathcal{D}$ (see Section 2). Thus, every component c^* can be identified and moved to next phase.

Remark. *If $r_c = r'$, this traversal can also be performed on a subtree from a component c of type $C2$ achieving similar result. This is possible because no new path p would be formed and we still get components of type $C1$ and $C2$ (being connected to a single path p_c).*

3.2 Path Halving

Consider a component of type $C2$ with $r_c \in p_c = path(x, y)$. We first find the farther end of p_c, say x, where $|path(r_c, x)| \geq |path(r_c, y)|$. We then traverse from r_c to x adding $path(r_c, x)$ to the tree T^* (see Figure 2 (b)). The component c' of type $C2$ thus formed will have $p_{c'}$ of length at most half of p_c. Now, the subtrees in c having an edge to $p_{c'}$ would be a part of c'. The remaining subtrees would form individual components of type $C1$. Again, the new root of each component can be found using $\mathcal{D}$ by querying for the lowest edge on the $path(r_c, x)$ added to T^*.

3.3 Disconnecting Traversal

Consider a component of type $C2$ with $r_c \in \tau$, where $\tau \notin \mathbb{T}_c$. We traverse τ from r_c to reach p_c, which is then followed by path halving of p_c. The goal is to ensure that the unvisited part of τ is not connected to the unvisited part of p_c (say p') after path halving, moving it to the next phase. The remaining subtrees of c with p' will move to the next stage as a result of path halving of p_c.

Now, if at least one edge from τ is present on the upper half of p_c, we find the highest edge from τ to p_c (see Figure 2 (c)). Otherwise, we find the lowest edge from τ to p_c. Let it be (x, y), where $y \in p_c$ and $x \in \tau$. This ensures that on entering p_c through y, path halving would ensure that all the edges from τ to p_c are incident on the traversed part of p_c (say p).

We perform the traversal from r_c to x similar to the *disintegrating traversal* along $path(r_c, x)$, attaching it to T^*. Since none of the components of unvisited part of τ are connected to p', all the components formed would be of type $C1$ or $C2$ as described in Section 3.1. However, while finding the new root of each resulting component c', we also need to consider the lowest edge from the component on p. Further, since $\tau \notin \mathbb{T}_c$, size of each subtree in the resulting components is at most $n/2^i$. Thus, the resultant components of τ are moved to the next phase.

Remark. *If $r_c \in T(v_H)$, this traversal can also be performed on a $\tau \in \mathbb{T}_c$ getting a similar result. This is because each subtree in resultant components of τ will have size at most $n/2^i$ moving it to the next phase. However, if $r_c \notin T(v_H)$ we cannot use this traversal as the resultant component c' of type $C2$ formed can have a heavy subtree and a path $p_{c'}$ of arbitrary length. This is not permitted as it will move the component to some earlier stage in the same phase. Hence, in such a case we would process the component using heavy subtree traversal described as follows.*

3.4 Heavy Subtree Traversal

Consider a component c of type $C2$ with $r_c \in \tau$, where $\tau \in \mathbb{T}_c$. As described earlier, if $r_c = root(\tau)$ or $r_c \in T(v_H)$, the heavy subtree τ can be processed using disintegrating or disconnecting traversals respectively. Otherwise, we traverse it using one of three scenarios described as follows. Our algorithm checks each scenario in turn for its applicability to τ, eventually choosing a scenario to perform an l, p or r traversal (see Figure 3). This traversal ensures that it is followed by a *simpler* traversal described earlier, so that each component will either move to the next phase or the next stage. We shall also prove that these scenarios are indeed exhaustive, i.e., for any τ, one of these scenarios is indeed applicable. The following lemma describes the conditions for a scenario to be applicable. Refer to the full paper [23] for proofs of the lemmas described in this section.

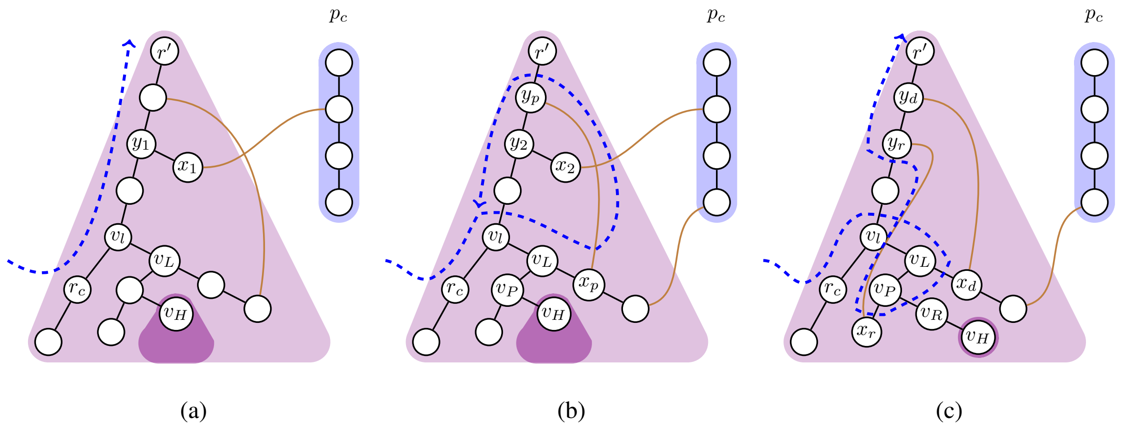

Figure 3: The three scenarios for Heavy Subtree Traversal (shown using blue dotted lines), (a) l traversal, (b) p traversal, and (c) r traversal.

Lemma 3.1 (Applicability Lemma). *After a traversal of path p^* in a subtree $\tau \in \mathbb{T}_c$, every component of unvisited part of c can be moved to the next phase/stage using a* simpler *traversal if*

- $\mathcal{A}_1$: *Traversal of p^* produces components of type C1 or C2 only,*
- $\mathcal{A}_2$: *The subtree $T(v_H)$ is connected to p_c (if in component of type C2),*
- $\mathcal{A}_3$: *The lowest edge on p^* from the component containing p_c is not a back edge from the subtree containing $T(v_H)$ with its end point outside $T(v_H)$.*

Remark. *Applicability lemma is employed when p^* does not traverse through $T(v_H)$. Otherwise, the unvisited component trivially moves to the next stage/phase This is because if p^* traverses through $T(v_H)$, the traversal preceding p^* was clearly applicable. Condition $\mathcal{A}_2$ ensures that the heavy subtree containing $T(v_H)$ does not form a component c' with arbitrary length of path $p_{c'}$, as this can move it to some previous stage which is not allowed. Because of the same reason* disconnecting *traversal is not used on such heavy subtrees.*

We now briefly describe the three scenarios, namely, l, p and r traversals and define a few notations related to them (shown in Figure 3). The l, p and r traversals follow the path shown in figure (using blue dotted lines) which shall henceforth be referred as p_L^*, p_P^* and p_R^* respectively. Both p and r traversals use a back edge during the traversal, denoted by (x_p, y_p) and (x_r, y_r) respectively. Further, we refer to the subtrees containing v_H that hangs from p_L^*, p_P^* and p_R^* as $T(v_L)$, $T(v_P)$ and $T(v_R)$ respectively. Any subtree hanging from the traversed path (p_L^*, p_P^* or p_R^*), shall be called an *eligible* subtree, if it has an edge to p_c. In each scenario we ensure $\mathcal{A}_1$ and $\mathcal{A}_2$ by construction, implying that the scenario will not be applicable only if $\mathcal{A}_3$ is violated. Thus, we only need to find the lowest edge on traversed path from the *eligible* subtrees, to determine the applicability of a scenario. Also, the edges (x_p, y_p) and (x_r, y_r) are chosen in such a way that if l and p traversals are not applicable, then r traversal always satisfies *applicability* lemma, with the lowest edge from component containing p_c being (x_d, y_d), where $x_d \in \tau_d \neq T(v_R)$.

Scenario 1: l traversal
Consider the traversal shown in Figure 3 (a), where $p_L^* = path(r_c, r')$. Since, this traversal does not create a new non-traversed path, $\mathcal{A}_1$ and $\mathcal{A}_2$ are implicitly satisfied. We find the lowest edge on p_L^* (highest edge on $path(r_c, r')$) from p_c and the *eligible* subtrees, say (x_1, y_1), where $y_1 \in p_L^*$. In case this edge satisfies $\mathcal{A}_3$, we perform the traversal otherwise move to the next scenario.

Remark. *This scenario is not applicable only if (x_1, y_1) is a back edge with $x_1 \in T(v_L)$ and $x_1 \notin T(v_H)$.*

Scenario 2: p traversal
Consider the traversal shown in Figure 3 (b), where $p_P^* = path(r_c, x_p) \cup (x_p, y_p) \cup path(y_p, par(v_l))$. To perform this traversal, we choose (x_p, y_p) along with (x_d, y_d) such that if p traversal using (x_p, y_p) is not applicable then r traversal using (x_d, y_d) is necessarily applicable. The presence of the back edge (x_1, y_1) (see remark of l traversal) can be used to show that (x_p, y_p) and (x_d, y_d) can be chosen satisfying the following properties Let τ_d and τ_p (if any), be the subtrees hanging from $path(v_L, v_H)$ containing x_d and x_p respectively.

Lemma 3.2. *The edge (x_p, y_p), which is a back edge, always exists and when used for p traversal satisfies $\mathcal{A}_1$ and $\mathcal{A}_2$.*

Lemma 3.3. *On performing r traversal using any edge (x_r, y_r), which satisfies (i) $x_r \in T(v_P) \cup \tau_p$ (if any), and (ii) the conditions $\mathcal{A}_1$ and $\mathcal{A}_2$, the r traversal is applicable with the lowest edge from an* eligible *subtree to p_R^* being (x_d, y_d) (from the* eligible *subtree τ_d), if either $\tau_d \neq \tau_p$ or τ_d is not traversed by p_R^*.*

Thus, Lemma 3.2 ensures that our traversal can follow p_P^* as shown in Figure 3 (b). To verify $\mathcal{A}_3$ we find the new root for the component having path p_c as follows. We find the lowest edge on p_P^* from p_c and the *eligible* subtrees hanging from p_P^*, say (x_2, y_2), where $y_2 \in p_P^*$. In case this edge satisfies $\mathcal{A}_3$, we perform the traversal otherwise move to the next scenario.

Remark. *This scenario is not applicable only if (x_2, y_2) is a back edge with $x_2 \in T(v_P)$ and $x_2 \notin T(v_H)$.*

Scenario 3: r traversal
Consider the traversal shown in Figure 3 (c), where $p^*_R = path(r_c, x_r) \cup (x_r, y_r) \cup path(y_r, r')$. We choose (x_2, y_2) as (x_r, y_r). However, while computing (x_2, y_2), τ_p (if exists) would have been partially traversed. Hence, if the lowest edge from τ_p to $path(r_c, r')$, say (x'_2, y'_2), has y'_2 lower than y_r on $path(r_c, r')$, τ_p would be connected to both p_c and $path(par(v_l), y_2) \backslash \{y_2\}$. This creates a component having p_c which is not of type C_1 or C_2 violating $\mathcal{A}_1$. In such a case we choose (x'_2, y'_2) as (x_r, y_r). The existence of back edge (x_2, y_2) (see remark of p traversal) implies the following property of (x_r, y_r).

Lemma 3.4. *The edge (x_r, y_r), which is a back edge, always exists and when used for r traversal satisfies $\mathcal{A}_1$ and $\mathcal{A}_2$.*

Thus, Lemma 3.4 ensures that our traversal can follow p^*_R as shown in Figure 3 (c). To verify $\mathcal{A}_3$ we find the new root for the component having path p_c as follows. We find the lowest edge on p^*_R from p_c and the *eligible* subtrees hanging from p^*_R, say (x_3, y_3), where $y_3 \in p^*_R$. In case this edge satisfies $\mathcal{A}_3$, we perform this traversal.

Using Lemma 3.3, we shall now describe the conditions when this edge satisfies the *applicability* lemma. The choice of (x_r, y_r) ensures that either $x_r \in T(v_P)$ (see remark in p traversal) or $x_r \in \tau_p$ (recall the computation of (x'_2, y'_2)). Further, using Lemma 3.4 our r traversal satisfies $\mathcal{A}_1$ and $\mathcal{A}_2$. Thus, using Lemma 3.3 the scenario is not applicable only in the *special case* where $\tau_d = \tau_p$ and p^*_R traverses τ_d.

We now present the conditions of the *special case* and present an overview of how it can be handled. Since $\tau_d = \tau_p$, p^*_R would traverse τ_d only if $x_r \in \tau_d = \tau_p$, i.e., $(x_r, y_r) = (x'_2, y'_2)$. Thus, both the lowest and the highest edges on $path(r_c, r')$, i.e., (x_p, y_p) and (x_r, y_r), from an eligible subtree hanging from $path(v_L, v_H)$ belong to τ_d. Moreover, since τ_d hangs from $path(v_L, v_H)$, it does not contain $T(v_H)$. This ensures that if modified r' traversal is performed ignoring τ_d, it can be followed by a disconnecting traversal of τ_d (see full paper [23] for details).

Correctness:

To prove the correctness of our algorithm, it is sufficient to prove two properties. *Firstly*, the components property is satisfied in each traversal mentioned above. *Secondly*, every component in a phase/stage, abides by the size constraints defining the phase/stage. By construction, we always choose the lowest edge from a component to the recently added path in T^* ensuring that the components property is satisfied. Furthermore, in different traversals we have clearly proved how each component progresses to the next stage/phase ensuring the size constraints. Thus, the final tree T^* returned by the algorithm is indeed a DFS tree of the updated graph.

Analysis

We now analyze a stage of the algorithm for processing a component c. In each stage, our algorithm performs at most $O(1)$ traversals of each type described above. Let us first consider the queries performed on the data structure $\mathcal{D}$. Every traversal described above performs $O(1)$ sets of these queries sequentially, where each set may have $O(|c|)$ parallel queries. Moreover, each of these sets is an *independent* set of parallel queries on $\mathcal{D}$ (recall the definition of *independent* queries in Section 2). This is because in each set of parallel queries, different queries are performed either on different untraversed subtrees of currently processed subtree or on the traversed path in the currently processed subtree. The remaining operations (excluding queries to $\mathcal{D}$) clearly requires only the knowledge of the current DFS tree T (and not whole G). Hence, they can be performed locally in the distributed and semi-streaming environment. Performing these operations efficiently in parallel shall be described in Section 4. Since our algorithm requires $\log n$ phases each having $\log n$ stages, we get the following theorem.

Theorem 3.5. *Given an undirected graph and its DFS tree T, any subtree τ of T can be rerooted at any vertex $r' \in \tau$ by sequentially performing $O(\log^2 n)$ sets of $O(|\tau|)$ independent queries on $\mathcal{D}$, in addition to local computation requiring only the subtree τ.*

4 IMPLEMENTATION ON EREW PRAM

We assign $|c|$ processors to process a component c, requiring overall n processors. We first present an efficient implementation of $\mathcal{D}$ and the operations on T used by our algorithm. These results are fairly simple or derived directly from classical results and hence may not be of independent interest (see full paper [23] for details).

Theorem 4.1. *The DFS tree T of a graph can be preprocessed to build a data structure $\mathcal{D}$ of size $O(m)$ in $O(\log n)$ time using m processors, such that a set of independent queries of types $Query(w, path(x,y))$, $Query(T(w), path(x,y))$ and $Query(path(v,w), path(x,y))$ on T can be answered simultaneously in $O(\log n)$ time using $1, |T(w)|$ and $|path(x,y)|$ processors respectively on an EREW PRAM.*

Remark. *Unlike the complicated $\mathcal{D}_0$ used by Baswana et al. [5], our data structure $\mathcal{D}$ is merely sorted adjacency lists of the graph according to the post order traversal of T.*

Theorem 4.2. *The DFS tree T of a graph can be preprocessed to build a data structure of size $O(n)$ in $O(\log n)$ time using n processors, such that the following queries can be answered in parallel in $O(\log n)$ time on an EREW PRAM*

- *LCA of two vertices, size of a subtree, testing if an edge is back edge and length of a path, using a single processor per query.*
- *Finding vertices on a path, subtrees hanging from a path, child subtree of a vertex containing a given vertex, highest/lowest edge among k edges, using k processors per query, where k is the size of the corresponding component.*

Using these data structures we can now analyze the time required by the *reduction* algorithm on an EREW PRAM. Since the queries on $\mathcal{D}$ and *LCA* queries on T can be answered in $O(\log n)$ time using n processors as described above, Theorem 2.2 reduces to the following theorem.

Theorem 4.3. *Given the DFS tree T of a graph and the data structure $\mathcal{D}$ built on it, any update on the graph can be reduced to independently rerooting disjoint subtrees of the DFS tree using n processors in $O(\log n)$ time on an EREW PRAM.*

Implementation details

Using Theorem 4.1 and Theorem 4.2, we can show that all operations required for each stage of our rerooting algorithm to reroot a subtree τ, can be performed in $O(\log n)$ time using $|\tau|$ processors. Both $root(\tau_c)$ and vertex v_H required by our algorithm while processing a component c, can be computed in parallel by comparing the size of each subtree using $|c|$ processors. Adding a path p to T^* essentially involves marking the corresponding edges as tree edges, which can be performed by informing the vertices on p. All the other operations of the rerooting algorithm are trivially reducible to the operations described in Theorem 4.2. Since our rerooting algorithm requires $\log n$ phases each having $\log n$ stages, we get the following theorem for rerooting disjoint subtrees using our rerooting algorithm.

Theorem 4.4. *Given an undirected graph with the data structure $\mathcal{D}$ build on its DFS tree, independently rerooting disjoint subtrees of the DFS tree can be performed in $O(\log^3 n)$ time using n processors on an EREW PRAM.*

Using Theorem 4.1 Theorem 4.3 and Theorem 4.4, we can prove our main result described as follows.

Theorem 1.1. *Given an undirected graph and its DFS tree, it can be preprocessed to build a data structure of size $O(m)$ in $O(\log n)$ time using m processors on an EREW PRAM, such that for any update in the graph, a DFS tree of the updated graph can be computed in $O(\log^3 n)$ time using n processors on an EREW PRAM.*

Now, in order to prove our result for Parallel Fully Dynamic DFS and Parallel Fault Tolerant DFS we need to first build the DFS tree of the original graph from scratch during the preprocessing stage. This can be done using the static DFS algorithm [37] or any advanced deterministic parallel algorithm [1, 17]. Thus, for processing any update we always have the current DFS tree built (either the original DFS tree built during preprocessing or the updated DFS tree built by our algorithm for the previous update). We can thus build the data structure $\mathcal{D}$ using Theorem 4.1 reducing Theorem 1.1 to the following theorem.

Theorem 4.5 (Parallel Fully Dynamic DFS). *Given an undirected graph, we can maintain its DFS tree under any arbitrary online sequence of vertex or edge updates in $O(\log^3 n)$ time per update using m processors on an EREW PRAM.*

However, if we limit the number of processors to n, our fully dynamic algorithm cannot update the DFS tree in $\tilde{O}(1)$ time, only because updating $\mathcal{D}$ in $\tilde{O}(1)$ time requires $O(m)$ processors (see Theorem 4.1). Thus, we build the data structure $\mathcal{D}$ using Theorem 4.1 during preprocessing itself, and attempt to use it to handle multiple updates.

Extending to multiple updates

Consider a sequence of k updates on graph, let T_i^* represent the DFS tree computed by our algorithm after i updates in the graph. We also denote the corresponding data structure $\mathcal{D}$ built on T_i^* as $\mathcal{D}_i$. Now, consider any stage of our algorithm while building the DFS tree T_i^*. For each component in parallel, $O(1)$ ancestor-descendant paths of T_{i-1}^* are added to T_i^*. Thus, any ancestor-descendant path p of T_i^*, is built by adding $O(\log^2 n)$ such paths of T_{i-1}^*, corresponding to $O(\log n)$ phases each having $O(\log n)$ stages. Hence, p is union of $O(\log^2 n)$ ancestor-descendant paths of T_{i-1}^*, say $p_1, ..., p_k$.

Using this reduction, it can be shown that a set of independent queries on path p in $\mathcal{D}_i$, can be reduced to $O(\log^2 n)$ sets of independent queries on corresponding $O(\log^2 n)$ paths $p_1, ..., p_k$ on $\mathcal{D}_{i-1}$ (see full paper [23] for details). Again, each of these paths $p_1, ..., p_k$, being an ancestor-descendant path of T_{i-1}^*, is a union of $O(\log^2 n)$ ancestor-descendant paths of T_{i-2}^*, and so on. Thus, any set of independent queries on $\mathcal{D}_i$ can be performed by $O(\log^{2(i-1)} n)$ sets of independent queries on $\mathcal{D}$, which takes $O(\log^{2i-1})$ time on an EREW PRAM using n processors when $k \leq \log n$ (see Theorem 4.1). The other data structures on T_{i-1}^* can be built in $O(\log n)$ time using n processors (see Theorem 4.2). This allows our algorithm to build the DFS tree T_i^* from T_{i-1}^* using $\mathcal{D}$ in $O(\log^{2i+1})$ time on an EREW PRAM using n processors (see Theorem 3.5). Thus, for a given set of k updates we build each T_i^* one by one using T_{i-1}^* and $\mathcal{D}$, to get the following theorem.

Theorem 4.6 (Parallel Fault Tolerant DFS). *Given an undirected graph, it can be preprocessed to build a data structure of size $O(m)$, such that for any set of k ($\leq \log n$) updates in the graph, a DFS tree of the updated graph can be computed in $O(k \log^{2k+1} n)$ time using n processors on an EREW PRAM.*

Remark. *For $k = 1$, our algorithm also gives an $O(n \log^3 n)$ time sequential algorithm for updating a DFS tree after a single update in the graph, achieving similar bounds as Baswana et al. [5]. However, our algorithm uses much simpler data structure $\mathcal{D}$ at the cost of a more complex algorithm.*

5 APPLICATIONS IN OTHER MODELS OF COMPUTATION

5.1 Semi-Streaming Setting

Our algorithm only stores the current DFS tree T and the partially built DFS tree T^* taking $O(n)$ space. Thus, all operations on T can be performed without any passes over the input graph. A set of independent queries on $\mathcal{D}$ is evaluated by performing a single pass over all the edges of the input graph using $O(n)$ space. This is because each set has $O(n)$ queries (see Theorem 2.2 and Theorem 3.5) and we are required to store only one edge per query (partial solution based on edges visited by the pass). Note that here the role of $\mathcal{D}$ is performed by a pass over the input graph. Hence, using Theorem 2.2 and Theorem 3.5 our algorithm requires $O(\log^2 n)$ passes per update proving our semi-streaming algorithm described as follows.

Theorem 5.1. *Given an undirected graph and its DFS tree, for any given update in the graph the updated DFS tree can be computed using $O(\log^2 n)$ passes over the input graph by a semi-streaming algorithm using $O(n)$ space.*

5.2 Distributed Setting

Our algorithm stores only the current DFS tree T and the partially built DFS tree T^* at each node. Thus, the operations on T are performed locally at each node and the distributed computation is only used to evaluate the queries on $\mathcal{D}$. Using Theorem 2.2 and Theorem 3.5, each update is performed by $O(\log^2 n)$ sequential sets of $O(n)$ independent queries on $\mathcal{D}$. Evaluation of a set of $O(n)$ independent queries on $\mathcal{D}$ can be essentially reduced to propagation of $O(n)$ words (partial solutions of n queries) throughout the network. Using the standard technique of pipelined broadcasts and convergecasts [30], we can propagate these $O(n)$ words in $O(D)$ rounds using messages of size $O(n/D)$. This proves our distributed algorithm described in Section 1.2. Refer to full paper [23] for details of implementation.

6 CONCLUSION

Our parallel dynamic algorithms take nearly optimal time on an EREW PRAM. However, the work efficiency of our fully dynamic algorithm is $\tilde{O}(m)$ whereas that of the best sequential algorithm [6] is $\tilde{O}(\sqrt{mn})$. Even though our fault tolerant algorithm is nearly work optimal, its only for constant number of updates. The primary reason behind these limitations is the difficulty in updating the data structure $\mathcal{D}$ using n processors. Our fault tolerant algorithm avoids this problem, by naively using the original $\mathcal{D}$ to simulate the queries of updated $\mathcal{D}$. It would be interesting to see if an algorithm can process significantly more updates using only n processors in $\tilde{O}(1)$ time (similar extension was performed by Baswana et al. [6] in the sequential setting). This may also lead to a fully dynamic algorithm that is nearly time optimal with better work efficiency.

Further, our distributed algorithm works only on a substantially restricted synchronous $\mathcal{CONGEST}(n/D)$ model. Moreover, the number of messages passed during an update in the distributed algorithm is $O(nD\log^2 n + m)$, which is way worse than the number of messages required to compute a DFS from scratch when the message size is relaxed, i.e., $O(n)$. It would be interesting to see if dynamic DFS can be maintained in near optimal rounds in more stronger $\mathcal{CONGEST}$ or $\mathcal{LOCAL}$ models.

ACKNOWLEDGMENTS

I am grateful to Christoph Lenzen for suggesting this problem. I would also like to express my sincere gratitude to my advisor Prof. Surender Baswana for valuable discussions and key insights that led to this paper. The idea of extending the parallel algorithm to the distributed setting was rooted in these discussions. Finally, I would like to thank the anonymous reviewers and my advisor whose critical assessment significantly helped me in refining the paper. This work is supported by Google India under the Google India PhD Fellowship Award.

REFERENCES

[1] Alok Aggarwal and Richard J. Anderson. 1988. A random NC algorithm for depth first search. *Combinatorica* 8, 1 (1988), 1–12.
[2] Alok Aggarwal, Richard J. Anderson, and Ming-Yang Kao. 1990. Parallel Depth-First Search in General Directed Graphs. *SIAM J. Comput.* 19, 2 (1990), 397–409.
[3] Sepehr Assadi, Sanjeev Khanna, Yang Li, and Grigory Yaroslavtsev. 2016. Maximum Matchings in Dynamic Graph Streams and the Simultaneous Communication Model. In *ACM-SIAM Symposium on Discrete Algorithms, SODA*. Society for Industrial and Applied Mathematics, Philadelphia, PA, USA, 1345–1364.
[4] Baruch Awerbuch, Israel Cidon, and Shay Kutten. 2008. Optimal maintenance of a spanning tree. *J. ACM* 55, 4, Article 18 (2008), 45 pages.
[5] Surender Baswana, Shreejit Ray Chaudhury, Keerti Choudhary, and Shahbaz Khan. 2015. Dynamic DFS Tree in Undirected Graphs: breaking the O(m) barrier, Full version of [6]. *CoRR* abs/1502.02481 (2015).
[6] Surender Baswana, Shreejit Ray Chaudhury, Keerti Choudhary, and Shahbaz Khan. 2016. Dynamic DFS in Undirected Graphs: breaking the O(m) barrier. In *ACM-SIAM Symposium on Discrete Algorithms, SODA*. Society for Industrial and Applied Mathematics, Philadelphia, PA, USA, 730–739.
[7] Surender Baswana and Keerti Choudhary. 2015. On Dynamic DFS Tree in Directed Graphs. In *Mathematical Foundations of Computer Science, MFCS*. Springer Berlin Heidelberg, Berlin, Heidelberg, 102–114.
[8] Surender Baswana and Shahbaz Khan. 2014. Incremental Algorithm for Maintaining DFS Tree for Undirected Graphs. In *ICALP*. Springer Berlin Heidelberg, Berlin, Heidelberg, 138–149.
[9] E. Boros, K. Elbassioni, V. Gurvich, and L. Khachiyan. 2000. An efficient incremental algorithm for generating all maximal independent sets in hypergraphs of bounded dimension. *Parallel Processing Letters* 10 (2000), 253–266.
[10] Israel Cidon. 1988. Yet Another Distributed Depth-First-Search Algorithm. *Inf. Process. Lett.* 26, 6 (1988), 301–305.
[11] Shimon Even and Robert Endre Tarjan. 1975. Network Flow and Testing Graph Connectivity. *SIAM J. Comput.* 4, 4 (1975), 507–518.
[12] Martin Farach-Colton, Tsan-sheng Hsu, Meng Li, and Meng-Tsung Tsai. 2015. Finding Articulation Points of Large Graphs in Linear Time. In *Algorithms and Data Structures, WADS*. Springer International Publishing, Cham, 363–372.
[13] P. Ferragina. 1995. A Technique to Speed Up Parallel Fully Dynamic Algorithms for MST. *J. Parallel Distrib. Comput.* 31, 2 (Dec. 1995), 181–189.
[14] Paolo Giulio Franciosa, Giorgio Gambosi, and Umberto Nanni. 1997. The Incremental Maintenance of a Depth-First-Search Tree in Directed Acyclic Graphs. *Inf. Process. Lett.* 61, 2 (1997), 113–120.
[15] Jon Freeman. October 1991. Parallel Algorithms for Depth-First Search. *Technical Report, University of Pennsylvania* (October 1991).
[16] Ratan K. Ghosh and G. P. Bhattacharjee. 1984. A Parallel Search Algorithm for Directed Acyclic Graphs. *BIT* 24, 2 (1984), 134–150.
[17] Andrew V. Goldberg, Serge A. Plotkin, and Pravin M. Vaidya. 1993. Sublinear-Time Parallel Algorithms for Matching and Related Problems. *J. Algorithms* 14, 2 (1993), 180–213.
[18] Sudipto Guha, Andrew McGregor, and David Tench. 2015. Vertex and Hyperedge Connectivity in Dynamic Graph Streams. In *ACM Symposium on Principles of Database Systems, PODS*. ACM, New York, NY, USA, 241–247.
[19] Torben Hagerup. 1990. Planar Depth-First Search in O(log n) Parallel Time. *SIAM J. Comput.* 19, 4 (1990), 678–704.
[20] Monika Henzinger, Sebastian Krinninger, and Danupon Nanongkai. 2013. Sublinear-Time Maintenance of Breadth-First Spanning Tree in Partially Dynamic Networks. In *ICALP (2)*. Springer-Verlag, Berlin, Heidelberg, 607–619.
[21] Zengfeng Huang and Pan Peng. 2016. Dynamic Graph Stream Algorithms in o(n) Space. In *43rd International Colloquium on Automata, Languages, and Programming (ICALP 2016)*. Schloss Dagstuhl–Leibniz-Zentrum fuer Informatik, Dagstuhl, Germany, 18:1–18:16.
[22] Junting Jin, Xiaowei Shi, Cuiping Li, and Hong Chen. 2014. Fast Approximation of Shortest Path on Dynamic Information Networks. In *Web-Age Information Management WAIM*. Springer International Publishing, Cham, 272–276.
[23] Shahbaz Khan. 2017. Near Optimal Parallel Algorithms for Dynamic DFS in Undirected Graphs. *CoRR* abs/1705.03637 (2017).
[24] Valerie King, Shay Kutten, and Mikkel Thorup. 2015. Construction and Impromptu Repair of an MST in a Distributed Network with o(m) Communication. In *Proceedings of the 2015 ACM Symposium on Principles of Distributed Computing, PODC*. ACM, New York, NY, USA, 71–80.
[25] Devendra Kumar, S. Sitharama Iyengar, and Mohan B. Sharma. 1990. Corrigenda: Corrections to a Distributed Depth-First Search Algorithm. *Inf. Process. Lett.* 35, 1 (1990), 55–56.
[26] K. B. Lakshmanan, N. Meenakshi, and Krishnaiyan Thulasiraman. 1987. A Time-Optimal Message-Efficient Distributed Algorithm for Depth-First-Search. *Inf. Process. Lett.* 25, 2 (1987), 103–109.
[27] S. A. M. Makki and George Havas. 1996. Distributed Algorithms for Depth-First Search. *Inf. Process. Lett.* 60, 1 (1996), 7–12.
[28] Andrew McGregor. 2014. Graph Stream Algorithms: A Survey. *SIGMOD Rec.* 43, 1 (May 2014), 9–20.
[29] Thomas C. O'Connell. 2009. A Survey of Graph Algorithms Under Extended Streaming Models of Computation. In *Fundamental Problems in Computing: Essays in Honor of Professor Daniel J. Rosenkrantz*, S. S. Ravi and Sandeep K. Shukla (Eds.). Springer Netherlands, Dordrecht, 455–476.
[30] David Peleg. 2000. *Distributed Computing: A Locality-sensitive Approach*. Society for Industrial and Applied Mathematics, Philadelphia, PA, USA.
[31] John H. Reif. 1985. Depth-First Search is Inherently Sequential. *Inf. Process. Lett.* 20, 5 (1985), 229–234.
[32] Jan Matthias Ruhl. 2003. Efficient Algorithms for New Computational Models. *PhD Thesis* Department of Computer Science, MIT, Cambridge, MA (2003).
[33] Kirk Schloegel, George Karypis, and Vipin Kumar. 2002. Parallel static and dynamic multi-constraint graph partitioning. *Concurrency and Computation: Practice and Experience* 14, 3 (2002), 219–240.
[34] Mohan B. Sharma and S. Sitharama Iyengar. 1989. An Efficient Distributed Depth-First-Search Algorithm. *Inf. Process. Lett.* 32, 4 (1989), 183–186.
[35] Deepak D. Sherlekar, Shaunak Pawagi, and I. V. Ramakrishnan. 1985. O(1) Parallel Time Incremental Graph Algorithms. In *Foundations of Software Technology and Theoretical Computer Science*. Springer Berlin Heidelberg, Berlin, Heidelberg, 477–495.

[36] Bala Swaminathan and Kenneth J. Goldman. 1998. An Incremental Distributed Algorithm for Computing Biconnected Components in Dynamic Graphs. *Algorithmica* 22, 3 (1998), 305–329.

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

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

[39] Robert Endre Tarjan. 1976. Edge-Disjoint Spanning Trees and Depth-First Search. *Acta Inf.* 6 (1976), 171–185.

[40] Yung H. Tsin. 2002. Some remarks on distributed depth-first search. *Inf. Process. Lett.* 82, 4 (2002), 173–178.

A REDUCTION ALGORITHM

We now describe how updating a DFS tree after any kind of update in the graph is equivalent to a simple procedure, i.e., *rerooting* disjoint subtrees of the DFS tree. Note that similar reduction was also used by Baswana et al. [5] but we describe it here for the sake of completeness as follows (see Figure 4).

(1) **Deletion of an edge** (u, v)**:**
If (u, v) is a back edge in T, simply delete it from the graph. Otherwise, let $u = par(v)$ in T. The algorithm finds the lowest edge (u', v') on the $path(u, r)$ from $T(v)$, where $v' \in T(v)$. The subtree $T(v)$ can then be rerooted to the new root v' and hanged from u' using (u', v') to get the final tree T^*.

(2) **Insertion of an edge** (u, v)**:**
In case (u, v) is a back edge, simply insert it in the graph. Otherwise, let w be the LCA of u and v in T and v' be the child of w such that $v \in T(v')$. The subtree $T(v')$ can then be rerooted to the new root v and hanged from u using (u, v) to get the final tree T^*.

(3) **Deletion of a vertex** u**:**
Let $v_1, ..., v_c$ be the children of u in T. For each subtree $T(v_i)$, the algorithm finds the lowest edge (u'_i, v'_i) on the $path(par(u), r)$ from $T(v_i)$, where $v'_i \in T(v_i)$. Each subtree $T(v_i)$ can then be rerooted to the new root v'_i and hanged from u'_i using (u'_i, v'_i) to get the final tree T^*.

(4) **Insertion of a vertex** u**:**
Let $v_1, ..., v_c$ be the neighbors of u in the graph. Arbitrarily choose a neighbor v_j and make u the child of v_j in T^*. For each v_i, such that $v_i \notin path(v_j, r)$, let $T(v'_i)$ be the subtree hanging from $path(v_j, r)$ such that $v_i \in T(v'_i)$. Each subtree $T(v'_i)$ can then be rerooted to the new root v_i and hanged from u using (u, v_i) to get the final tree T^*.

In case of a vertex update, multiple subtrees may be required to be rerooted by the algorithm. Let these subtrees be $T_1, ..., T_c$. Notice that each of these subtrees can be rerooted independent of each other, and hence in parallel. However, in order to perform the *reduction* algorithm efficiently in parallel, we require a structure to answer the following queries efficiently in parallel. (a) Finding LCA of two vertices in T. (b) Finding the highest edge from a subtree $T(v)$ to a path in T (a query on data structure $\mathcal{D}$). In addition to these we also require several other types of queries to be efficiently answered in parallel setting as testing if an edge is back edge, finding vertices on a path, child subtree of a vertex containing a given vertex etc. However, these can easily be answered using LCA queries (see full paper [23] for details). Thus, we have the following theorem.

THEOREM 2.2. *Given an undirected graph G and its DFS tree T, any graph update can be reduced to independently rerooting disjoint subtrees of T by performing $O(1)$ sets of independent queries on the data structure $\mathcal{D}$ and $O(1)$ sets of LCA queries on T, where each set has at most n queries.*

REMARK. *The implementation of reduction algorithm is simpler in distributed and semi-streaming environments, where any operation on the DFS tree T can be performed locally without any distributed computation or passes over the input graph respectively. Hence, for these environments the reduction algorithm requires only $O(1)$ sets of independent queries on the data structure $\mathcal{D}$.*

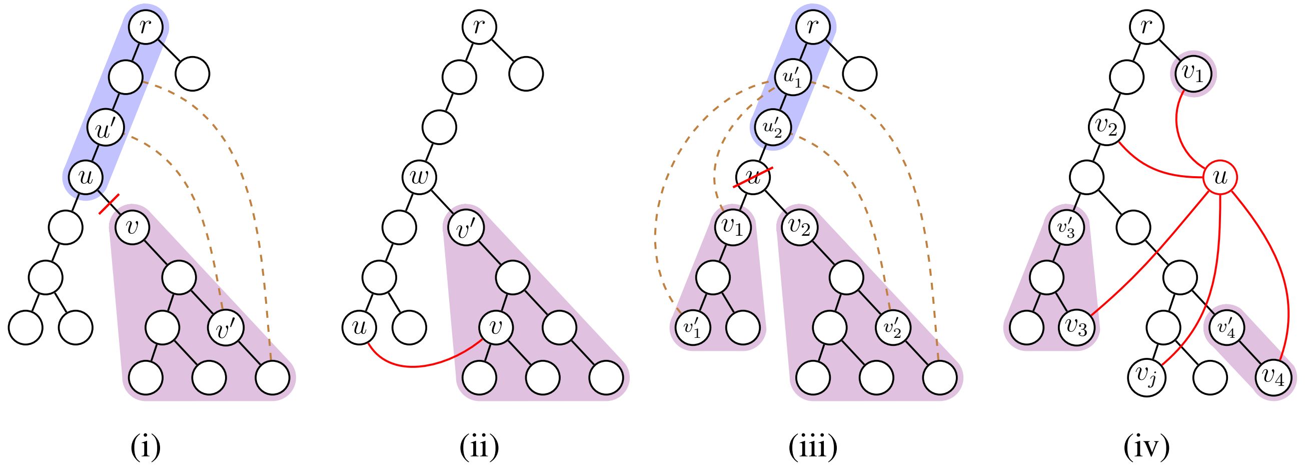

Figure 4: Updating the DFS tree after a single update: (i) deletion of an edge, (ii) insertion of an edge, (iii) deletion of a vertex, and (iv) insertion of a vertex. The reduction algorithm reroots the marked subtrees (shown in violet) and hangs it from the inserted edge (in case of insertion) or the lowest edge (in case of deletion) on the marked path (shown in blue) from the marked subtree (reproduced from [5]).