Depth First Search Trees in Dynamic Graphs

A Thesis Submitted

in Partial Fulfillment of the Requirements

for the Degree of

Doctor of Philosophy

by

Shahbaz Khan

to the

DEPARTMENT OF COMPUTER SCIENCE AND ENGINEERING

INDIAN INSTITUTE OF TECHNOLOGY KANPUR

September, 2017

CERTIFICATE

It is certified that the work contained in the thesis entitled "Depth First Search Trees in Dynamic Graphs", by *Shahbaz Khan*, has been carried out under my supervision, and that this work has not been submitted elsewhere for a degree.

Surender Baswana
Department of Computer Science and Engineering
Indian Institute of Technology Kanpur

September, 2017

Abstract

In most of the graph applications in the real world, we deal with graphs that undergo structural changes in the form of insertion or deletion of vertices or edges. For any arbitrary online sequence of graph updates, a dynamic graph algorithm has to report the solution of the problem after every update in the graph. This can trivially be achieved by using the best static algorithm to recompute the solution from scratch after each update. Thus, the objective is to update the solution of the problem much faster than the time required to recompute the solution from scratch. A dynamic graph algorithm is called *fully dynamic* if it handles both insertion and deletion updates. A *partially dynamic* algorithm handles either only insertion updates (*incremental*) or only deletion updates (*decremental*). Another model of practical significance is the *fault tolerant model*, where the aim is to build a compact data structure for a given problem, which can be used to compute the solution efficiently after any set of edge or vertex failures. Our work focuses on dynamic graph algorithms for maintaining Depth First Search trees, wherein we developed both theoretically and empirically efficient algorithms.

Depth First Search (DFS) tree is one of the most popular graph data structures having innumerable applications in solving graph problems. Despite its undeniable significance in the area of graph algorithms, its surprisingly limited impact in dynamic graph problems seemed unsettling. The only non-trivial algorithm then known, was by Franciosa et al. [IPL97] dating back to mid 90s, which was restricted to incremental setting in directed acyclic graphs. Recently, Baswana and Choudhary [MFCS15] presented a randomized decremental algorithm for directed acyclic graphs. Moreover, most of the problems that are trivially solvable by a DFS tree in the static setting, did not employ any form of DFS tree in the dynamic setting. Several such problems including connectivity, reachability, strong connectivity and their variants, are some of the most studied problems in the area of dynamic graph algorithms.

We first studied the problem of maintaining a DFS tree in the incremental setting. We developed an extremely simple algorithm [ICALP14] for incrementally maintaining a DFS tree of an undirected graph. Our algorithm is essentially optimal for dense graphs.

However, in all the existing algorithms for maintaining a DFS tree in the dynamic setting, only amortized guarantees in the update time were achieved. In the worst case, the time required to update the solution was no better than the time required to recompute the solution from scratch. Thus, in all the existing solutions an inherent difficulty to achieve a better worst case update time was noticed. We were able to break this barrier for undirected graphs [SODA16] in fault tolerant, incremental and fully dynamic settings. Our fault tolerant (up to constant failures) and incremental algorithms require nearly optimal time (up to $poly \log n$ factors) to update the DFS tree of the graph. Moreover, our fully dynamic algorithm also opened the gateways for applicability of DFS trees in other well studied dynamic graph problems such as dynamic subgraph connectivity and its variants. Our focus then shifted to the study of dynamic DFS in other more practical models of computation, where we were able to design nearly optimal algorithms (up to $poly \log n$ factors) for maintaining a DFS tree of an undirected graph in the parallel, semi-streaming and distributed model [SPAA17] Finally, we carried out an exhaustive experimental and theoretical evaluation [SODA18] of the existing algorithms for maintaining incremental DFS in random graphs and real world graphs. This led us to develop extremely simple algorithms which performed both theoretically equivalent and experimentally superior to the existing algorithms on dense random graphs. Even for real graphs, we propose new algorithms that match the performance of the state-of-the-art algorithms. Finally, we present tradeoffs to choose the most suitable algorithm for any given dynamic graph.

Hence, we are able to successfully address the problem of dynamic DFS from several directions: ranging from theoretical to experimental, sequential to parallel/distributed/semi-streaming, and near optimal amortized to significant worst case guarantees.

Synopsis

Name of Student	**Shahbaz Khan**
Roll Number	13111165
Degree for which submitted	Doctor of Philosophy(PhD)
Department	Department of Computer Science and Engineering, Indian Institute of Technology Kanpur
Thesis Title	Depth First Search Trees in Dynamic Graphs
Supervisor	Surender Baswana
Submission Date	September, 2017

In most of the graph applications in the real world, we deal with graphs that undergo structural changes in the form of insertion or deletion of vertices or edges. For any arbitrary online sequence of graph updates, a dynamic graph algorithm has to report the solution of the problem after every update in the graph. This can trivially be achieved by using the best static algorithm to recompute the solution from scratch after each update. Thus, the objective is to update the solution of the problem much faster than the time required to recompute the solution from scratch. A dynamic graph algorithm is called *fully dynamic* if it handles both insertion and deletion updates. A *partially dynamic* algorithm handles either only insertion updates (*incremental*) or only deletion updates (*decremental*). Another model of practical significance is the *fault tolerant model*, where the aim is to build a compact data structure for a given problem, which can be used to compute the solution efficiently after any set of edge or vertex failures.

Depth First Search (DFS) tree is one of the most popular graph data structures having innumerable applications in solving graph problems. Despite its undeniable significance in the area of graph algorithms, its surprisingly limited impact in dynamic graph problems seemed unsettling. The only non-trivial algorithm then known, was by Franciosa et al. [FGN97] dating back to mid 90s, which was restricted to incremental setting in directed acyclic graphs. Recently, Baswana and Choudhary [BC15] presented a randomized decremental algorithm for directed acyclic graphs achieving similar bounds (up to $\tilde{O}(1)$ factors)[1] as in [FGN97]. Moreover, most of the problems that are trivially solvable by a

[1] $\tilde{O}()$: Hides the factors poly-logarithmic in input size, i.e., $O(Xpoly\log n) = \tilde{O}(X)$ for input size n.

DFS tree in the static setting, did not employ any form of DFS tree in the dynamic setting. Several such problems including connectivity, reachability, strong connectivity and their variants, are some of the most studied problems in the area of dynamic graph algorithms.

In this thesis, we study Depth First Search trees in the dynamic setting, wherein we developed both theoretically and empirically efficient algorithms. Our contributions can be broadly classified in the following domains.

Dynamic DFS in Undirected Graphs

As described earlier, in any dynamic setting, a trivial algorithm can use the classical static algorithm [Tar72] to compute the DFS tree from scratch after every update. We improve this trivial algorithm in the following dynamic environments.

Incremental Algorithms

Franciosa et al. [FGN97] stated the maintenance of a DFS tree incrementally in an undirected graph as an open problem. We presented an incremental algorithm [BK14, BK17] to maintain a DFS tree of an undirected graph under edge insertions, which is optimal for dense graphs. Moreover, it is also very simple and employs only basic data structures, making it ideal for practical use.

Despite having significant amortized bounds, in the worst case, none of these algorithms guarantee an update time any better than the trivial recomputation from scratch. Our later work [BCCK16] presented an algorithm for incrementally maintaining a DFS tree for an undirected graph in nearly optimal (up to $\tilde{O}(1)$ factors) worst case update time.

Fault tolerant Algorithm

In the fault tolerant model, no non-trivial algorithm was previously known for DFS trees. We presented a fault tolerant algorithm [BCCK16] to recompute the DFS tree after any k failures in the graph. Moreover, it also handles vertex updates in addition to edge updates using the same bounds. For small values of k (up to $\tilde{O}(1)$), this algorithm is also shown to be optimal up to $\tilde{O}(1)$ factors.

Fully Dynamic Algorithm

In the fully dynamic setting, again no non-trivial algorithm was previously known for maintaining a DFS tree. Further, the dynamic DFS algorithms described above did not have any applications in solving dynamic graph problems, unlike DFS in the static setting.

We presented an algorithm [BCCK16] to maintain a DFS tree for any sequence of edge or vertex updates in a fully dynamic environment. Moreover, our algorithm also seamlessly provides new, simple, and efficient algorithms for several well studied dynamic graph problems such as connectivity [Dua10, CPR11], biconnectivity [Hen00], and 2-edge connectivity [HdLT01] in the dynamic subgraph model. Our result improves the deterministic worst case bounds for these problems under vertex updates. Recall that in the static setting these problems are easily solvable using a DFS tree.

Extensions to other models of computation

Major applications of dynamic graphs in the real world involve a huge amount of data, which not only makes recomputing the solution after every update infeasible, but also solving it on a single sequential machine impractical. Thus, in the past three decades a lot of work addressed dynamic graph problems on more practical computation models as parallel, semi-streaming and distributed (or dynamic networks) models. However, despite its significance, DFS trees have only been studied on these models in the static setting.

Parallel Model

Aggarwal et al. [AAK90] proved that general DFS tree problem is in RNC[2]. However, the fastest deterministic algorithm for computing general DFS tree in parallel takes polynomial time [AAK90, GPV93] even for undirected graphs. Whether the DFS tree problem is in NC, is still a long standing open problem even for undirected graphs.

We presented the first parallel dynamic algorithms [Kha17a] for maintaining a DFS tree for undirected graphs, which exponentially improves the deterministic update time over trivial re-computation after every update. Our fully dynamic algorithm maintains a DFS tree in $\tilde{O}(1)$ time per edge/vertex update using one processor per edge of the graph. Further, using only one processor per vertex of the graph, our fault tolerant algorithm computes a DFS tree after any set of k edge/vertex updates in $\tilde{O}(1)$ time (for constant k). Both our algorithms are time optimal (up to $\tilde{O}(1)$ factors) and also establish that dynamic DFS for undirected graphs is in class NC.

Streaming Model

In the semi-streaming model, the input graph is accessed in the form of a stream of edges over multiple passes, where the algorithm is allowed only linear space. A DFS tree can

[2] NC is the class of problems solvable using n^c processors in parallel in $\tilde{O}(1)$ time, for any constant c. The class RNC extends NC to allow access to randomness.

be trivially computed using one pass per vertex, where the pass adds the vertex to the tree. However, computing the DFS tree in $\tilde{O}(1)$ passes is considered hard [FHLT15] and it remains an open problem to even compute it in sublinear number of passes.

In the dynamic setting, after every graph update the algorithm is allowed multiple passes over the graph to update the DFS tree. We presented the first dynamic semi-streaming algorithm [Kha17a] for maintaining a DFS tree of an undirected graph, which exponentially improves the number of passes required over trivial re-computation after every update. Our algorithm maintains a fully dynamic DFS tree using $\tilde{O}(1)$ passes over the input graph for every edge/vertex update, which is optimal (up to $\tilde{O}(1)$ factors).

Distributed model

Computing a DFS tree in the distributed model was widely studied in 1980's and 1990's. A DFS tree can be computed in linear number of rounds, with different tradeoffs between the number of messages passed and the size of each message.

We presented an algorithm [Kha17a] which maintains a DFS tree of an undirected graph after any edge/vertex update using rounds of the order of the diameter of the graph. Despite using linear message size with higher number of messages, it improves the number of rounds required for the classes of graphs with sublinear diameter.

Empirical analysis of Incremental DFS algorithms

Having known several algorithms for maintaining incremental DFS [FGN97][BK17, BCCK16], for practical use it is necessary to evaluate their average performance and perform their empirical analysis on real inputs. After all, the ideal goal is to design an algorithm with theoretical guarantee of efficiency as well as efficient performance in practice. Thus, such a study bridges the gap between theory and practice.

We presented an experimental analysis [BGK18] of various algorithms for incremental DFS. Both our algorithm [BK17] and that of Franciosa et al. [FGN97], perform much better than their stated bounds on random graphs. We investigated the reasons behind this superior performance and theoretically proved probabilistic bounds which almost matched (up to $\tilde{O}(1)$ factors) their empirical performance on dense random graphs. Using this insight, we presented an extremely simple algorithm for incremental DFS that works for both undirected and directed graphs. This algorithm theoretically matches and experimentally outperforms the state-of-the-art algorithms in dense random graphs. Further, it can also be used as a single-pass semi-streaming algorithm for incremental DFS and strong connectivity for random graphs.

Even for real graphs, both our algorithm [BK17] and that of Franciosa et al. [FGN97], perform much better than their stated bounds. Here again, we propose new algorithms that match the performance of the state-of-the-art algorithms. Finally, we present trade-offs to choose the most suitable algorithm for any given dynamic graph.

Hence, we are able to successfully address the problem of dynamic DFS from several directions: ranging from theoretical to experimental, sequential to parallel/distributed/semi-streaming, and near optimal amortized to significant worst case guarantees.

To my alma mater and my family...

Acknowledgements

I am extremely grateful to my advisor Prof. Surender Baswana who has been a constant source of support, motivation and guidance throughout my PhD. It was particularly the work I did with him during my Masters which inspired me to pursue research in Algorithms and undertake the daunting task of doctoral studies. His immense love and appreciation to the area, always motivates me to find more meaning and beautiful insights behind the pale looking algorithms. It was only because of such a mentorship that I am now able to connect better to the ideas and appreciate the approaches taken by various researchers of the field. He has shown utmost care and vigilance towards my work, giving me ample freedom as well as constant reminders and close introspection whenever I fell off track. His concerns about the timely progress of my PhD have always been much more than that of my own, where he also guided me in advance for various opportunities and challenges beyond my doctoral studies. Our long discussions were especially enlightening wherein we were able to feed off each others ideas and promptly develop examples to counter incorrect approaches. It has been truly an honour to work with not just a great supervisor and an excellent researcher, but a renowned teacher whose attitude towards the responsibility as a teacher has inspired in me utmost respect towards this noble service. Above all he has been a very enthusiastic, hard working and supportive mentor, who has treated me not merely as an advisee but as a naive child, nurturing me in every aspect of this experience and constantly guiding me to develop into a mature researcher. I may not be ever able to match his expectations given the amount of time, effort and care he invested in me, but would always remember it as the motivation behind pursuing excellence in every aspect of my career ahead.

I would also like to particularly thank the faculty and staff at the Department of CSE, IIT Kanpur for creating an environment of research and learning, shaping every aspect of my life here. I particularly cherish the courses I undertook with Prof. Surender Baswana, Prof. Shashank Mehta, Prof. Sanjeev Saxena, Prof. Satyadev Nandakumar, Prof. Nitin Saxena and Prof. Sumit Ganguly. They were not only an inspiration towards learning the subject, but also challenged our grasp on every concept revealing its insightful applications.

During my PhD I had the opportunity of several academic visits which have truly been helpful in exposing me to various opportunities for research and collaboration. I would like to express my sincere gratitude to Google India for awarding me Google India PhD Fellowship in Algorithms, whose flexible and generous funding have made possible a lot of such visits. I also thank Prof. Liam Roditty for hosting me at Bar Ilan University, Israel and UGC-ISF Grant for funding the trip. I would also like to thank the Algorithms and Complexity Group at Max Planck Institute of Informatics for hosting me in MPII Saarbrucken, Germany and IMPECS project for funding the trip. I also thank Sayan Bhattacharya and Manoj Gupta for the opportunity of research collaboration during this visit. I would like to additionally thank Manoj Gupta for the opportunity of a prosperous collaboration and for hosting me during my visit to IIT Gandhinagar.

My life at IIT Kanpur would not have been such an enjoyable experience without the help and support from all my friends at IIT Kanpur. Most of all I am indebted to Tejas Gandhi for being a close friend and mentor through the entire journey, who have always helped and supported me in all the lows and highs of my life here. The experimental work that I completed as a part of thesis would not have been possible without the help and expert guidance of Tejas Gandhi and Siddharth Kumar Rai. I also particularly cherish the innumerable discussions with my lab mates at KD 308, which brought peace and sanity to our lives amidst various phases of our lives here.

I am ever grateful to our faculty in Department of Computer Engineering at my alma mater Aligarh Muslim University. The core foundations, technical knowledge and invaluable guidance I attained from them have allowed me to cruise through this journey as a PhD student much easier. I would also like to particularly thank Dr. Saiful Islam and my senior Abed Mohammad Kamaluddin whose invaluable support and guidance motivated me to pursue post graduate and doctoral studies.

Lastly and most significantly, I would like to thank my family for shaping me into whatever I am today, and in supporting and encouraging me throughout this journey. I would like to particularly thank my brother and best friend Shahzor Khan, who have always believed in me, perhaps more than myself, and guided me to take this path. He has truly played an invaluable role to guide and motivate me in this and every other part of my life. Above all, I am the most thankful to the Almighty God, for honouring me with faith which has been a constant motivation to pursue excellence in every aspect of my life.

Contents

List of Publications

[BCCK15] Surender Baswana, Shreejit Ray Chaudhury, Keerti Choudhary, and Shahbaz Khan. Dynamic DFS tree in undirected graphs: breaking the O(m) barrier, full version of [BCCK16]. *CoRR*, abs/1502.02481, 2015.

[BCCK16] 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.

[BGK17] Surender Baswana, Ayush Goel, and Shahbaz Khan. Incremental DFS algorithms: a theoretical and experimental study. *CoRR*, abs/1705.02613, 2017.

[BGK18] Surender Baswana, Ayush Goel, and Shahbaz Khan. Incremental DFS algorithms: a theoretical and experimental study. In *Proceedings of the Twenty-Ninth Annual ACM-SIAM Symposium on Discrete Algorithms, SODA 2018, New Orleans, LA, USA, January 7-10, 2018*, pages 53–72, 2018.

[BK14] Surender Baswana and Shahbaz Khan. Incremental algorithm for maintaining DFS tree for undirected graphs. In *ICALP*, pages 138–149, 2014.

[BK17] Surender Baswana and Shahbaz Khan. Incremental algorithm for maintaining a DFS tree for undirected graphs. *Algorithmica*, 79(2):466–483, 2017.

[Kha17a] 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.

[Kha17b] Shahbaz Khan. Near Optimal Parallel Algorithms for Dynamic DFS in Undirected Graphs. *CoRR*, abs/1705.03637, 2017.

Contents of Chapter 2 is based on the results of [BK14] whose full version is available in [BK17]. Chapter 3 is based on [BCCK16] with full version available in [BCCK15]. The work in Chapter 4 is based on [Kha17a] having the full version in [Kha17b]. Finally, Chapter 5 is based on [BGK18] with the full version in [BGK17].

List of Algorithms

List of Figures

List of Tables

Chapter 1

Introduction

Most of the data in the real world that deals with complex relationships, is represented using Graphs. Various algorithms are thus designed to process these graphs and answer various queries on them. These queries vary from merely answering if two nodes are connected (*connectivity*), to measuring the shortest distance/path between two nodes (*shortest paths*), to even building complex structures for these graphs including minimum weight spanning trees (*minimum spanning trees*), maximum concurrent transport routes (*maximum flow*) etc. If the underlying input graph is not subjected to any change while processing these queries, the corresponding algorithms are referred as *static* algorithms. On the other hand, if the underlying input graph is subject to any changes, the corresponding algorithms are referred as *dynamic* algorithms. In this thesis, we develop dynamic graph algorithms for one of the most fundamental data structures that is used for solving graph problems in the static setting: the Depth First Search trees. We shall now formalize the notion of dynamic graph algorithms and briefly describe the motivation, goals, and the state of the art for various graphs problems studied under this model.

1.1 Dynamic Graph Algorithms

Most of the graph applications in real world deal with graphs that keep changing with time. These changes/updates can be in the form of insertion or deletion of vertices or edges. An algorithmic graph problem is modeled in a dynamic environment as follows. For any arbitrary online sequence of updates on the graph, the objective is to update the solution of the problem efficiently after each update. To achieve this aim, we maintain some clever data structure for the problem on the current graph, such that the *update time*, i.e., time taken to compute the solution after an update in the graph, is much smaller than the time required by the best static algorithm to compute the solution from scratch.

Dynamic graph algorithms also have significant impact in other areas of theoretical interests. They are often used as an intermediate routine in solving some graph problems in the static setting. A classical example is that of Kruskal's algorithm [Kru56] for computing Minimum Spanning Trees, which uses a disjoint set union data structure. It is essentially a data structure for answering connectivity queries under insertion of edges. Moreover, the complexity of dynamic graph problems are also known to be strongly related to the complexity of the corresponding problems in the parallel setting. In the mid 80s and early 90s, several studies [Rei87, MSVT94] reported the relationship between the hardness of a problem in the parallel and dynamic settings.

1.1.1 Types of Dynamic Graph Algorithms

We now describe the different types of dynamic graphs algorithms. A dynamic graph algorithm is said to be *fully dynamic* if it handles both insertion as well as deletion updates. A *partially dynamic* algorithm is said to be *incremental* or *decremental* if it handles only insertion or only deletion updates respectively.

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 failure of vertices/edges, and can efficiently report the solution of the problem for any given set of failures. Typically the size of this set of failures is assumed to be much smaller than the number of vertices or edges in the graph. In this model, *update time* often refers to the processing time required by the data structure after a given set of failures, such that each query can be answered efficiently.

1.1.2 State of the art

We now briefly describe some important problems studied in the dynamic environment.

Connectivity

Dynamic connectivity is one of the most studied problem in the area of dynamic graph algorithms, which can be described as follows. Given an undirected graph $G = (V, E)$ undergoing updates, the goal is to efficiently answer the following query: For any $u, v \in V$, is u connected to v in the graph G?

In the *fully dynamic* setting, the best known deterministic algorithm with worst case guarantees requires $O(\sqrt{n})$ time per update with $O(1)$ query time. This result was an improvement over Frederickson's [Fre85] $O(\sqrt{m})$ update time data structure, using the *sparsification* technique by Eppstein et al. [EGIN97]. Recently, this result was mildly improved by Kejlberg-Rasmussen et al. [KKPT16] to require $O(\sqrt{n\frac{(\log\log n)^2}{\log n}})$ update time.

However, there do exist faster algorithms if we settle for amortized guarantees or allow randomization. In a seminal paper, Henzinger and King [HK99] presented an algorithm to maintain fully dynamic connectivity in expected $O(\log^3 n)$ amortized update time with $O(\log n/\log\log n)$ query time. This was later improved to expected $O(\log n(\log\log n)^3)$ update time with $O(\log n/\log\log\log n)$ query time by Thorup [Tho00], and to expected $O(\log n(\log\log n)^2)$ update time with same query time by Huang et al. [HHKP17]. On the other hand, the best deterministic algorithm with amortized guarantees is by Holm et al. [HdLT01] which takes $O(\log^2 n)$ amortized update time and $O(\log n/\log\log n)$ query time. This was later improved by Wulff-Nilsen [Wul13] to $O(\log^2 n/\log\log n)$ update time with same query time. Allowing randomization also improves the worst case bound of $O(\sqrt{n})$. In another seminal paper, Kapron et al. [KKM13] presented a Monte-Carlo algorithm[1] requiring $O(\log^5 n)$ update time which correctly answers the queries with high probability in $O(\log n/\log\log n)$ time, assuming an *oblivious adversary* [2]. Recently, Nanongkai and Saranurak [NS17] presented a Monte-Carlo algorithm for *adaptive adversary* [3] re-

[1] A randomized algorithm in which answers to the queries can be incorrect with some probability, but have deterministic guarantee on the running time is referred as a Monte-Carlo algorithm.

[2] An *oblivious* adversary does not have access to the previous random choices made by the algorithm.

[3] An *adaptive* adversary can adapt based on the previous random choices made by the algorithm.

quiring expected $O(n^{0.4+o(1)})$ update time. They also presented a Las Vegas algorithm[4] requiring $O(n^{0.49306})$ worst case update time with high probability. Independently, Wulff-Nilsen [Wul17] also presented a Las Vegas algorithm requiring $O(n^{0.5-c})$ worst case update time with high probability, for some constant $c > 0$.

In case we restrict ourselves to *partially dynamic* environment, we can achieve better bounds as follows. In the *incremental* setting, the disjoint set union data structure [Tar75, TvL84] can be used to answer the connectivity queries taking total $O(m\alpha(m, n))$ time for queries and insertion of m edges, where $\alpha(m, n)$ is the inverse Ackermann's function. In the *decremental* setting, the only known algorithm which performs better than the corresponding fully dynamic algorithm is by Thorup [Tho99]. This algorithm requires expected $O(m \log(n^2/m)) + n \log^3 n(\log \log n)^2)$ time for deletion of m edges, where each query can be answered in $O(1)$ time. This bound is better than that of [Tho00] for $m = \Omega(n \log^2 n)$.

Apart from these upper bounds, a lot of work has also been done to prove lower bounds for the problem. For incrementally maintaining connectivity information, Tarjan [Tar79] and La Poutré [Pou96] independently proved lower bounds of $\Omega(\alpha(m, n))$ per update on a pointer machine. Henzinger and Fredman [HF98] established a lower bound for fully dynamic connectivity requiring $\Omega(\log n/ \log \log n)$ time per update in the RAM model [5]. This was later improved by Demaine and Patrascu [PD06] to $\Omega(\log n)$ time per update in the cell-probe model[6].

In the *fault tolerant* setting, the algorithms for fully dynamic connectivity having worst case bounds can also be used for solving fault tolerant connectivity. Additionally, Patrascu and Thorup [PT07] presented a fault tolerant connectivity structure which takes $O(k \log^{5/2} n \log \log n)$ update time after any set of k edge failures to answer each connectivity query in $O(\log \log n)$, using polynomial preprocessing time. This result was improved by Duan and Pettie [DP10] for $k = o(\log^{5/3} n)$, requiring $O(k^2 \log \log n)$ update time using $O(\min\{\frac{\log k}{\log \log n}, \frac{\log \log n}{\log \log \log n}\})$ query time. They also presented a vertex fault tolerant connectivity structure for any $k \leq k_{max}$ failures, which offers a tradeoff between update time and size of the structure with a query time of $O(k)$. For a constant parameter c, their data structure requires $O(k_{max}^{1-2/c} m n^{1/c-1/(c \log 2k_{max})} \log^2 n)$ space and $O(k^{2c+4} \log^2 n \log \log n)$ update time. They recently improved this result [DP17] to $O(mk_{max} \log n)$ space and $O(d^3 \log^3 n)$ update time. Further, allowing randomization they also presented a Monte Carlo algorithm improving the space to $O(m \log^4 n)$ and the update time $O(d^2 \log^5 n)$.

Some other problems that are closely related to connectivity are minimum spanning trees, 2-edge connectivity and biconnectivity. Many of the results for dynamic connectivity are easily extendible to minimum spanning trees [Fre85, EGIN97, HdLT01, NS17, Wul17], 2-edge connectivity [Fre85, EGIN97, HK99, Tho99, Tho00, HdLT01] and biconnectivity [HK99, Tho00, HdLT01] at the expense of extra $O(poly \log n)$ factors in the update time. Additionally, for maintaining fully dynamic minimum spanning forest, Holm et al. [HRW15] mildly improved the amortized update time of [HdLT01] from $O(\log^4 n)$ to $O(\log^4 n \log \log \log n/ \log \log n)$. For dynamic biconnectivity, Henzinger [Hen00] also presented a fully dynamic algorithm that requires $O(\sqrt{n})$ update time and $O(1)$ query time.

[4] A randomized algorithm which always answers the queries correctly, but has only expected or high probability guarantees on the running time is referred as a Las Vegas algorithm.

[5] Random Access Machine (RAM) model allows random access of memory locations and integer operations in constant time.

[6] Cell probe model measures computational cost only in terms of the number of memory cells accessed.

Another model in which dynamic connectivity has been studied is the dynamic subgraph model described as follows. Given an undirected graph, where the status of any vertex can be switched between *active* and *inactive* in an update. For any online sequence of updates interspersed with queries, the goal is to efficiently answer each queries on the subgraph induced by the active vertices. This problem can be solved by using fully dynamic data structures that answer the corresponding queries under an online sequence of edge updates. This is because switching the state of a vertex is equivalent to $O(n)$ edge updates. Chan [Cha06] introduced this problem and showed that it can be solved using FMM (fast matrix multiplication) in $O(m^{0.94})$ amortized update time and $\tilde{O}(m^{1/3})$ query time. Later, Chan et al. [CPR08] presented a new algorithm that improves the amortized update time to $\tilde{O}(m^{2/3})$. Further, they also established a lower bound of $\Omega(\sqrt{m})$ on the update time. Duan [Dua10] presented an algorithm with $O(m^{4/5})$ update time and $O(m^{1/5})$ query time, improving the worst case bounds for the problem. Recently, Duan and Zhang [DZ17] presented a randomized Monte-Carlo algorithm requires $\tilde{O}(m^{3/4})$ update time and answers a query correctly in $\tilde{O}(m^{1/4})$ time with high probability. Interestingly, most of these algorithm demonstrate a tradeoff between the update time and query time, where the product of the two parameters as $\Omega(m)$. Henzinger et al. [HKNS15] proved a matching conditional lower bound on this product using the OMv conjecture [7].

Reachability

For directed graphs, the problem equivalent to dynamic connectivity is dynamic reachability, which can be described as follows. Given a graph $G = (V, E)$ undergoing updates, the goal is to efficiently answer the following query: For any $u, v \in V$, is v reachable from u? In other words, does there exist a path from u to v in the graph G? The problem is addressed in two forms, namely *reachability*, from a single source; and *transitive closure*, among all possible pairs of vertices. See [DI06a, Mar17] for surveys.

In the *fully dynamic* setting, transitive closure can be maintained by a deterministic algorithm using $O(n^2)$ amortized update time and a constant query time. The first such algorithm was designed by Demetrescu and Italiano [DI08]. Later, Roditty [Rod08] presented a simpler algorithm having same query and update time with a smaller preprocessing time. For a data structure maintaining a reachability matrix this is optimal, since a single update may change $O(n^2)$ entries of the matrix. For randomized algorithms, Sankowski [San04] later improved it to worst case $O(n^2)$ update time with constant query time using a Monte-Carlo algorithm.

However, if we allow non-constant query time, there are different tradeoffs between update time and query time described as follows. Sankowski [San04] generalized the results of [DI05] to present two Monte Carlo algorithms requiring $O(n^{1.575})$ update time with $O(n^{0.575})$ query time, and $O(n^{1.495})$ update with $O(n^{1.495})$ query time respectively. Roditty and Zwick [RZ08] improved the results of [HK95], to present a deterministic algorithm having $O(m\sqrt{n})$ amortized update time with worst case $O(\sqrt{n})$ query time, and a randomized Las Vegas algorithm requiring amortized $O(m^{0.58}n)$ update time and worst case $O(m^{0.43})$ update time with high probability. Later, Roditty and Zwick [RZ16] also presented a deterministic algorithm requiring amortized $O(m + n \log n)$ update time and worst case $O(n)$ query time.

[7] Online Matrix Vector (OMv) multiplication conjecture states that no algorithm can compute a sequence of multiplications of the form $M.v$, of an online sequence of n vectors v $(1 \times n)$ with a fixed matrix M of size $n \times n$, in $O(n^{3-\epsilon})$ total time for any $\epsilon > 0$

Again, restricting to *partially dynamic* environments we get better bounds as follows. Single source reachability can be trivially maintained in the *incremental* setting using $O(1)$ amortized time per update [Ita86]. In case of transitive closure, correspondingly the incremental algorithm takes $O(n)$ amortized bound per update [Ita86, LPvL88]. For bounded degree graphs, this bound was improved to $O(md)$ total time by Yellin [Yel93], where d is the maximum out degree of a vertex in the final graph. In the *decremental* setting, single source reachability can be maintained by the classical result of Even and Shiloach [ES81] requiring $O(n)$ amortized update time. However, if we allow randomization decremental reachability can be maintained using a Monte Carlo algorithm by Chechik et al. [CHI$^+$16] which requires $O(\sqrt{n}\log n)$ amortized update time, improving upon [HKN14b, HKN15]. For transitive closure, the fastest algorithm is by Lacki [Lac13] requiring $O(n)$ amortized time per update with constant query time, improving upon [HK95, RZ08]. However, in the worst case [Lac13] may require $O(mn)$ time per update, which was improved by Roditty [Rod13] to $O(m \log n)$ keeping the same total update time as [Lac13]. All these algorithms require constant query time.

A related problem is that of maintaining dynamic strongly connected components , and answering strong connectivity queries, which are described as follows: Given a directed graph, for any two vertices $u, v \in V$, do they belong to the same strongly connected component. Many algorithms for dynamic reachability [Lac13, RZ08, HKN14b, HKN15, CHI$^+$16] also maintain strongly connected components allowing strong connectivity queries using the same bounds.

The long history of upper bounds for this problem is also complemented by some interesting lower bounds. Abboud and Williams [AW14] establishes a conditional lower bound implying the impossibility of a combinatorial algorithm solving decremental $s-t$ reachability with worst case $O(n^{2-\epsilon})$ (for any $\epsilon > 0$) query and update time, using BMM conjecture[8]. Removing the restriction of being a combinatorial algorithm, Henzinger et al. [HKN15] showed that no worst case $O(n^{1-\epsilon})$ update time and $O(n^{2-\epsilon})$ query time algorithm is possible under the OMv Conjecture.

In the *fault tolerant* setting, the only known data structures that handle multiple failures is by Baswana et al. [BCR16, BCR17]. For answering single source reachability and strong connectivity queries [BCR16] in $O(1)$ time, their algorithm require $O(2^k n)$ update time after k edge/vertex failures in the graph. For reporting the strongly connected components of a graph after k edge/vertex failures their algorithm [BCR17] requires $O(2^k n \log^2 n)$ time.

Shortest Paths

Dynamic maintenance of shortest paths is another problem that has been extensively studied over the past decades. Given a graph $G = (V, E)$ undergoing updates, the goal is to efficiently answer the following query: For any $s, t \in V$, what is the shortest path from s to t in G? In case the algorithm allows such queries among all pairs of vertices in V, the problem is referred to as all pairs shortest paths (APSP). However, if the query restricts the s to be a fixed vertex in V, the problem is called single source shortest path (SSSP). Further, if the graph G is unweighted, the shortest path tree from a given source is also called as the breadth first search (BFS) tree of the graph. See [DI06a, Mar17] for surveys.

[8]Boolean Matrix Multiplication (BMM) conjecture states that no combinatorial algorithm can compute multiplication of two $n \times n$ matrices using $O(n^{3-\epsilon})$ time, for any $\epsilon > 0$

In the *fully dynamic* setting, maintenance of APSP is first studied by King [Kin99], who presented an algorithm requiring $\tilde{O}(n^{2.5}\sqrt{W})$ amortized update time, where W is the maximum weight of an edge. This was improved to nearly quadratic amortized update time of $\tilde{O}(n^2 \log^3 n)$ time in the seminal work by Demetrescu and Italiano [DI04], which was later improved by a logarithmic factor by Thorup [Tho04]. Further, Thorup [Tho05] presented an algorithm having worst update time of $\tilde{O}(n^{2.75})$, which was recently improved by some logarithmic factors by Abraham et al. [ACK17]. They also improved the bound using randomization to give a Monte-Carlo algorithm with worst case update time of $\tilde{O}(n^{2+1/3})$. If we allow non-constant query time, Roditty and Zwick [RZ11] also presented a Monte-Carlo algorithm requiring $\tilde{O}(m\sqrt{n})$ amortized update time with $O(n^{3/4})$ query time. Using Fast Matrix Multiplication (FMM), Sankowski [San05] presented a Las Vegas algorithm requiring worst case $O(n^{1.932})$ update time and $O(n^{1.288})$ query time. Several results have also addressed the approximate maintenance of APSP [Kin99, Ber09].

In the *incremental* setting, the fastest APSP algorithm is by Ausiello et al. [AIMN91] requiring total $\tilde{O}(n^3W)$ update time under edge insertions , where W is the largest edge weight. In the decremental setting, the classical algorithm by Even and Shiloach [ES81] provides an algorithm requiring $O(mn^2)$ total update time. This was improved by Demetrescu and Italiano using randomization giving two algorithms requiring total $\tilde{O}(n^3)$ and $\tilde{O}(n^3S)$ update time respectively for vertex [DI06b] and edge updates [DI04], where S is the number of different edge weights. For maintenance of SSSP in partially dynamic environment, the fastest algorithm is the classical result by Even and Shiloach [ES81] requiring total update time of $O(mn)$ for unweighted graphs. For positively weighted graphs, King [Kin99] extended the result requiring total update time of $O(mnW)$, where W is the maximum weight of an edge. Later, Roditty and Zwick [RZ11] proved a matching conditional lower bound to [ES81] for any combinatorial algorithm maintaining partially dynamic SSSP using APSP conjecture [9]. This conditional lower bound was later generalized to even non-combinatorial algorithms by Henzinger et al. [HKNS15] using OMv conjecture. Subsequently, several results studied the partially dynamic approximate maintenance of SSSP in [BR11, HKN14a, BC17b, Ber17] and APSP in [Ber16, HKN16].

In the *fault tolerant* model, data structures for distance queries were first studied by Demetrescu et al. [DTCR08], handling a single failure. Later, its preprocessing time was improved [BK09], and it was extended to dual failures [DP09]. All these algorithms require $\tilde{O}(n^2)$ space and $\tilde{O}(1)$ query time. It was generalized to handle k failures using Fast Matrix Multiplication (FMM) resulting in a super-quadratic sized data structure by Weimann and Yuster [WY13]. Later, their sub-quadratic query time was improved to sublinear by Grandoni and Williams [GW12]. For SSSP, fault tolerant subgraphs have been studied which preserve the shortest paths from the given source after a set of failures. For weighted graphs, it was shown [DTCR08] that for even a single failure we require $\Omega(n^2)$ edges. However, for unweighted graphs (BFS trees) Parter and Peleg [PP13] presented the construction of a sub-quadratic sized fault tolerant subgraph, avoiding a single failure with a matching lower bound. This was extended to dual failures [Par15, GK17], again with matching lower bounds. Further, Parter [Par15] proved a lower bound of $\Omega(n^{2-\frac{1}{k+1}})$ edges for a k fault tolerant subgraph. Recently, construction of the first k fault tolerant subgraph was presented by Bodwin et al. [BGPW17], requiring $O(n^{2-\frac{1}{2^k}})$ edges.

[9] All pairs shortest paths (APSP) conjecture states that no algorithm can compute all pair shortest paths in $O(n^{3-\epsilon})$ total time for any $\epsilon > 0$

1.2 Other Models of Computation

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 involved. Several such models considered include parallel model, semi-streaming model and distributed model as follows.

1.2.1 Parallel Model

Parallel model of computation explores the use of multiple processors to work together in parallel in solving the given problem often using a shared memory. The popular shared memory model [FW78] assumes a global shared memory accessed by multiple processors, which are parallel random access machines (PRAM), i.e., are allowed to access the shared memory in parallel and perform integer operations in constant time. Based on criteria for simultaneous read/write access by multiple processors on the same memory cell, this model is classified in three types. Exclusive Read Exclusive Write (EREW) model restricts any two processors to simultaneously read or write the same memory cell. Concurrent Read Exclusive Write (CREW) model relaxes this restriction for reading, allowing any number of processors to simultaneously read the same memory cell. Concurrent Read Concurrent Write (CRCW) model further relaxes this restriction to writing as well.

In the past three decades a lot of work has been done to address dynamic graph problems in parallel environment. Several significant graph problems are studied in parallel dynamic setting including k-Connectivity [PR86, FL96, LS95, LBS01], connected components [PR86, FL96], shortest paths [LMS96] and graph partitioning [OR97]. However, the most extensively studied problem in this model is minimum spanning trees (MST).

Pawagi and Ramakrishnan [PR86] presented the first parallel dynamic algorithms for maintaining MST under vertex insertion or edge insertion/deletion in $O(\log n)$ time per update using n^2 processors on a CREW PRAM. This improved over recomputation from scratch after an update using the best static algorithm requiring $O(\log^2 n)$ time. Tsin [Tsi88] extended it to handle vertex deletions using similar bounds. Later, Varman and Doshi [VD86] improved the total work done per update for maintaining MST under vertex insertions, reducing the processors required to n. This was further improved to $n/\log n$ processors by Jung and Melhorn [JM88] but for a more relaxed CRCW PRAM model. Further, Pawagi and Kaser [PK93] improved the bounds for edge update and deletion of a vertex of degree d, to $O(\log n)$ and $O(\log n + \log^2 d)$ update time respectively using $O(n^2/\log n)$ processors on a CREW PRAM. The vertex deletion bound was improved for high degree vertices by Shen and Laing [SL93], requiring $O(\log n \cdot \log d)$ update time using $O(n^2/\log n \log d)$ processors. Similarly, the edge update bound was improved for sparse graphs by Johnson and Metexas [JM96] requiring $O(\log n)$ update time using $O(m/\log n)$ processors on a stricter EREW model. Ferragina and Luccio [FL96] improved the work efficiency of the algorithm at the cost of higher update time of $O(\log^{3/2} n \log\log \frac{m}{n})$ using $O(n \log \frac{m}{n} / \log\log \frac{m}{n})$ processors. Das and Ferragina[DF99] further improved it by presenting two algorithms requiring $O(\log n)$ update time using $O(m^{2/3}/\log n)$ processors, and $O(\log n \log \frac{m}{n})$ update time using $O(n^{2/3}/\log n)$ processors. Finally, it was improved to $O(\log n)$ update time using $O(n^{2/3} \log \frac{m}{n} / \log n)$ processors by Ferragina [Fer95].

Maintaining MST have also been studied under batch updates. Pawagi [Paw89] first studied the problem for a batch of k vertex insertions, requiring $O(\log n \log k)$ update time using $O(nk)$ processors on a CREW PRAM. Pawagi and Kaser [PK93] improved the work efficiency of the problem allowing both k vertex insertion or k edge insertions (or weight decrease) in $O(\log n \cdot \log k)$ update time using $O(nk/\log n \log k)$ processors on a CREW PRAM. They also showed k edge cost increase/deletion in $O(\log n + \log^2 d)$ update time using $O(n^2/\log n)$ processors CREW. The k vertex insertion result was also proved for the stricter EREW PRAM model by Johnson and Metexas [JM96]. Shen and Laing [SL93] improved the number of processors required for k edge insertions and deletions in $O(\log n \log k)$ update time to $O(n(1 + \frac{k}{\log n \log k}))$ and $O(\frac{n^2}{\log n \log k})$ respectively. Finally, Ferragina and Luccio [FL96] presented two algorithm for batch edge insertions and deletions requiring $O(\log^{\frac{3}{2}} n)$ update time using $O(n \log \frac{m}{n})$ processors, and $O(\log^{\frac{3}{2}} n \log \frac{m}{n})$ update time using $O(\min\{bn, m\})$ processors respectively on a CREW PRAM.

1.2.2 Semi-Streaming Model

Streaming model [AMS99, FKSV03, GKS01] is a popular model for computation on large data sets. In this model the input data is accessed as a stream, typically in a single pass over the input, allowing very small storage space (*poly* log in input size). For most graph problems such limited space was found to be impractical which led to proposal of relaxed streaming models as the *semi-streaming model* [Mut05, FKM$^+$05], allowing $\tilde{O}(n)$ space.

Dynamic graph updates in the semi-streaming model have been studied using three popular models: *insert-only* model, *insert-delete* model and *sliding window* model. In the *insert-only* model, we have a stream of edges that are inserted in the graph. This is similar to the static semi-streaming model, where all the edges of the graph are accessed in the form of an input stream using a single pass. The *insert-delete* model [AGM12a] considers a stream of edges that are either inserted or deleted from the graph. In the *sliding window* model [CMS13] we have a stream of edges, where only the recent k edges are considered. Ideally, the aim is to develop single pass algorithms but several results even consider constant or $O(\log n)$ number of passes. Refer to [McG14, McG17] for surveys.

The study of dynamic streams in the *insert-delete* model was initiated by Ahn et al. [AGM12a]. They used *graph sketches* to develop efficient algorithms for testing graph connectivity, k connectivity, bipartiteness, finding minimum spanning trees and cut sparsifiers. The result was later extended to report a witness for k edge connectivity [AGM12b] and approximately testing k vertex connectivity [GMT15] using $\tilde{O}(nk)$ space. The cut sparsifier by Ahn et al. [AGM12a] required $O(\log n)$ passes, which was later improved to a single pass [AGM12b, GKP12]. Spectral sparsifiers were first studied in this model by Ahn et al. [AGM13] requiring super-linear space. This was later improved to linear space requiring two passes by Kapralov and Woodruff [KW14], and finally to a single pass by Kapralov et al. [KLM$^+$17]. Approximate maximum matching is known to be computable using $\tilde{\Theta}(n^2/\alpha^3)$ space [AKLY16, CCE$^+$16], where α is the approximation factor. This reduces to $\tilde{O}(n^2/\alpha^4)$ if only the size of matching is reported [AKL17]. Further, for reporting the maximum matching of size k the required space is $\tilde{\Theta}(k^2)$ [BS15, CCE$^+$16]. Other significant problems considered in this model include finding densest subgraphs [BHNT15, MTVV15, ELS15, EHW16] and counting triangles [MVV16, BC17a, KP17]. Additionally, there have also been some progress in proving lower bounds for problems in the dynamic semi-streaming models [SW15, AKLY15, Kon15].

1.2.3 Distributed Model

Distributed model for computation is widely studied as it closely simulates the real world networks. Popular models of distributed computation for solving graph problems include $\mathcal{CONGEST}$ and $\mathcal{LOCAL}$ models [Pel00]. In both these models, computing nodes having local storage are available at each vertex of the graph, where the edges of the graph represent the communication channels between the nodes. The communication between two nodes takes place by passing messages in synchronous/asynchronous rounds. In the $\mathcal{CONGEST}$ model, the size of a message is typically bounded by $O(\log n)$ bits (or $O(1)$ words). A relaxed variant of this model is $\mathcal{CONGEST}(B)$ which relaxes the message size to $O(B)$ words. In the $\mathcal{LOCAL}$ model the size of a message is typically unbounded, however the local storage space at each node is limited to $O(\log n)$ size.

Dynamic graphs in the distributed model (also called *dynamic networks*) have been studied since late 70s (see [ACK08] for a brief survey). Most of the earlier works focussed on efficiently performing a *reset* of the earlier computation to recompute the solution from scratch [AAG87, AS88, Gaf87, SG89]. In a seminal work, Awerbuch et al. [ACK08] demonstrated the maintenance of a spanning tree using amortized $O(n)$ messages improving from $O(m)$ messages required for recomputation from scratch. Some of the most studied problems in this model includes the maintenance of all pair shortest paths (APSP), breadth first search (BFS) trees and minimum spanning trees (MST).

Ramrao and Venkatesan [RV92] presented a fully dynamic and an incremental algorithm for maintaining APSP requiring $O(n^3)$ messages and $O(n^2)$ messages respectively per update, using $O(n)$ space. For incremental all pairs shortest paths, Italiano [Ita91] improved the number of messages to $O(n \log nw)$, where w is the maximum weight of an edge. The problem is also studied [CSFN03, CDSF10] in terms of output sensitivity (number of vertex pairs whose shortest paths are updated), to yield faster results if output sensitivity is $o(n^2)$.

BFS trees can be maintained in the incremental and decremental setting by extending the classical result of Even and Shiloach [ES81], requiring $O(nD)$ total update time, where D is the diameter of the graph. Henzinger et al. [HKN13] improved the result for dense graph at the expense of approximating distances by $(1+\epsilon)$ factor, requiring total update time of $O(n^{1/3}(Dq/\epsilon)^{2/3})$ and $O(n^{1/5}(Dq)^{4/5}/\epsilon)$ respectively for insertion and deletion of q edges/vertices. Ghaffari and Parter [GP16] presented the first algorithm to compute fault tolerant BFS structure, using $O(D \log n)$ rounds.

Distributed construction of fault tolerant MST structures was first studied by Flocchini et al. [FEP$^+$12], who presented a structure resilient to an edge/vertex failure which can be constructed in $O(n)$ rounds using $O(n^2)$ messages. Other results study fast retrieval of MST [DLPP13] and reporting a spanning tree having minimum diameter [GSW11], after an edge failure. Recently, Ghaffari and Parter [GP16] improved the construction time of fault tolerant BFS tree to $O(D + n\sqrt{n})$ rounds. Further, King et al. [KKT15] recently improved the worst case message complexity of rebuilding the MST after an edge insertion or deletion to $O(n)$ and $\tilde{O}(n)$ respectively.

Other significant problems studied in this setting include colouring [LEK08], maximal independent sets [CHHK16], spanners [Elk07, BKS12], biconnected components [SG98, PPC02] and 3 edge connectivity [Jen97].

1.3 Experimental Analysis

For various graph algorithms [Mey01, ALM96, BMST06], the average-case time complexity (average performance on random graphs) has been proven to be much less than their worst case complexity. A classical example is the algorithm by Micali and Vazirani [MV80] for maximum matching. Its average case complexity has been proved to be only $O(m \log n)$ [Mot94, BMST06], despite having a worst case complexity of $O(m\sqrt{n})$. An equally important aspect is the empirical performance of an algorithm on real world graphs. After all, the ideal goal is to design an algorithm having a theoretical guarantee of efficiency in the worst case as well as superior performance on real graphs. Often such an experimental analysis also leads to the design of simpler algorithms that are extremely efficient in real world applications. The algorithm by Micali and Vazirani [MV80] has also been empirically analysed [MR91, Cro91, KP98, HS17] resulting in several important heuristics to improve its performance on various types of graphs. Thus, such an analysis bridges the gap between theory and practice. The experimental analysis of different algorithms for several dynamic graph problems has been performed as follows (also see [Zar00] for an exhaustive survey).

Dynamic connectivity algorithms were first evaluated by Alberts et al. [ACI97a], who studied the performance of sparsification [EGIN97] based algorithms and the randomized algorithm by Henzinger and King[HK99]. This study was extended by Fatourou et al. [FSZZ99] who additionally evaluated another randomized algorithm by Nikoletseas et al. [NRSY95]. Later, Iyer et al. [IKRT01] investigated these algorithms with the deterministic algorithm by Holm et al. [HdLT01].

The problem of maintaining dynamic minimum spanning trees was first investigated by Amato et al. [ACI97b], wherein they compared the effect of sparsification [EGIN97] on the data structure by Frederickson [Fre85] along with other algorithms. Later, Ribeiro and Toso [RT07] studied the problem under changing edge weights comparing various approaches based on dynamic trees. Finally, Cattaneo et al. [CFPI10] compared these algorithms with the algorithm by Holm et al. [HdLT01] with some simple algorithms which performed well in practice.

Dynamic transitive closure was first investigated by Abdeddaim [Abd00] which studied various partially dynamic (incremental) algorithms. Later, Frigioni et al. [FMNZ01] additionally evaluated several fully dynamic algorithms including that of Henzinger and King [HK95] and Italiano [Ita86]. Finally, Krommidas and Zaroliagis [KZ08] extended the study by additionally evaluating several other algorithms [DI05, Kin99, KT01, Rod08, RZ08, RZ16].

Perhaps the most exhaustively evaluated dynamic graph problem is the maintenance of shortest paths. Frigioni et al. [FINP98] investigated the significance of the dynamic algorithms over static algorithms in practice for single source shortest paths with non-negative edge weights. Later, Demetrescu et al. [DFMN00] generalized this study to allow arbitrary edge weights. Further, Buriol et al. [BRT08] investigated the impact of *heap reduction* on such algorithms. Additionally, a few studies have also evaluated the problem under batch updates [BW09, DDF$^+$15], only updating edge weights [DW07, SS07] and maintaining all pair shortest paths [DFIT06, DI06b].

Other dynamic problems of practical significance which have been evaluated experimentally include dominators [Sre95, PGT11, GILS12], topological sorting [PK06] and dynamic maximum flows [KT07, KT08, GHK$^+$15], etc.

1.4 Depth First Search

Depth First Search (DFS) is a well known graph traversal technique. This technique has been reported to be introduced by Charles Pierre Trémaux, a 19th-century French mathematician who used it for solving mazes. However, it was Tarjan, who in his seminal work [Tar72], demonstrated the power of DFS traversal for solving various fundamental graph problems, namely, topological sorting, connected components, biconnected components and strongly-connected components. Since then, DFS traversal is one of the most widely used graph traversal techniques having played the central role in the design of efficient algorithms for many other graph problems including bipartite matching [HK73], dominators in directed graph [Tar74], planarity testing [HT74], edge and vertex connectivity [ET75] etc. Interestingly, the role of DFS traversal is not confined to merely the design of efficient algorithms. For example, consider the classical result of Erdős and Rényi [ER60] for the phase transition phenomena in random graphs. There exist many proofs of this result which are intricate and based on highly sophisticated probability tools. However, recently, Krivelevich and Sudakov [KS13] designed a truly simple, short, and elegant proof for this result based on the insights from a DFS traversal in a graph.

Let $G = (V, E)$ be an undirected connected graph having n vertices and m edges. The DFS traversal of G produces a rooted spanning tree (or forest), called DFS tree (or forest), in $O(m+n)$ time. For any ordered rooted spanning tree, the non-tree edges of the graph can be classified into four categories as follows. An edge directed from a vertex to its ancestor in the tree is called a *back edge*. Similarly, an edge directed from a vertex to its descendant in the tree is called a *forward edge* . Further, an edge directed from right to left in the tree is called a *cross edge*. The remaining edges directed from left to right in the tree are called *anti-cross edges* . A necessary and sufficient condition for such a tree to be a DFS tree is the absence of anti-cross edges. In case of undirected graphs, this condition reduces to the absence of all cross edges. 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.

In spite of the simplicity of a DFS tree, designing any efficient parallel or dynamic algorithm for a DFS tree has turned out to be quite challenging. [Rei85, Rei87] was the first to address the complexity of the DFS problem in a dynamic environment. He showed [Rei85] that the ordered DFS tree problem is a P-Complete problem. Reif [Rei87] and later Miltersen et al. [MSVT94] proved that P-Completeness of a problem also implies hardness of the problem in the dynamic setting. The work of Miltersen et al. [MSVT94] shows that if the ordered DFS tree is updateable in $O(poly \log n)$ time, then the solution of every problem in class P is updateable in $O(poly \log n)$ time. In other words, maintaining the ordered DFS tree is indeed the hardest among all the problems in class P. In our view, this hardness result, which is actually for only the ordered DFS tree problem, has proved to be quite discouraging for the researchers working in the area of dynamic algorithms. This is evident from the fact that for all the static graph problems that were solved using DFS traversal in the 1970's, none of their dynamic counterparts used a dynamic DFS tree [HK99, HdLT01, KKM13, RZ08, Cha06, CPR08, Dua10].

1.4.1 Dynamic algorithms for DFS

Apart from showing the hardness of the ordered DFS tree problem, very little work has been done on the design of any non-trivial algorithm for the problem of maintaining any DFS tree in a dynamic environment. For the case of a directed acyclic graph (DAG), Franciosa et al. [FGN97] presented an incremental algorithm for maintaining a DFS tree in $O(mn)$ total time. Recently, again only for a DAG, Baswana and Choudhary [BC15] presented a decremental algorithm for maintaining a DFS tree in expected $O(mn \log n)$ total time. These are the only non-trivial results available for the dynamic DFS tree problem.

Maintaining a DFS tree incrementally for an undirected graph (or general directed graph) was stated as an open problem by Franciosa et al. [FGN97]. These algorithms are the only results known for the dynamic DFS tree problem. Moreover, none of these existing algorithms, though designed for only a partially dynamic environment, achieves a worst case bound of $o(m)$ on the update time. Furthermore, none of these results proves that general DFS is not as hard as ordered DFS in the dynamic environment. This is because the speculations of having to incur a complete recomputation in the worst case after an update is not disproved by amortized bounds resulting in the perceived $O(m)$ *barrier* for general DFS as well.

1.4.2 DFS in other models of computation

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. Moreover, in each of these models DFS have only been addressed in the static setting (except a minor result [Cha90] for parallel), which can be described as follows.

Parallel Algorithms

As described earlier, Reif [Rei85] showed the hardness of the ordered DFS tree problem in the parallel setting by proving that it is a P-Complete problem. For many years, this result seemed to imply that the general DFS tree problem, that is, the computation of any DFS tree of the graph is also inherently sequential. However, Aggarwal et al. [AA88, AAK90] proved that the general DFS tree problem is in RNC [10] by designing a randomized EREW PRAM algorithm that takes $\tilde{O}(1)$ expected time. But the fastest deterministic algorithm for computing general DFS tree in parallel still takes $\tilde{O}(\sqrt{n})$ time [AAK90, GPV93] in CRCW PRAM [11] , even for undirected graphs. Moreover, the general DFS tree problem has been shown to be in NC for some special graphs including DAGs [GB84, Zha86, KC86] and planar graphs [Hag90, Kao88, Smi86] (see [Fre91] for a survey). For DAGs, Chaudhuri [Cha90] presented the only parallel dynamic algorithm for maintaining a DFS tree, requiring slightly better update time (by $O(\log d)$ factor) than recomputing using a parallel static algorithm [KC86] after an edge/vertex insertion, where d is the diameter of

[10] NC is the class of problems solvable using $O(n^{c_1})$ processors in parallel in $O(\log^{c_2} n)$ time, for any constants c_1 and c_2. The class RNC extends NC to allow access to randomness.

[11] 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.

the DAG. In fact for random graphs in $G(n, p)$ model [12] [ER59], Dyer and Frieze [DF91] proved that even ordered DFS tree problem is in *RNC*. Whether general DFS tree problem is in *NC* is still a long standing open problem.

Streaming Algorithms

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 over the input graph stream, where each pass adds one vertex to the DFS tree. However, computing the DFS tree in $\tilde{O}(1)$ passes is considered hard [FHLT15]. To the best of our knowledge, it remains an open problem to compute a DFS tree using even $o(n)$ passes in any relaxed streaming environment [O'C09, Ruh03].

Distributed Algorithms

Computing a DFS tree in a distributed setting was widely studied in 1980's and 1990's. A DFS tree of the given graph can be computed in $O(n)$ rounds, with different trade offs between number of messages passed, and size of each message. If the size of a message is allowed to be $O(n)$, the DFS tree can be built using $O(n)$ messages [KIS90, MH96, SI89]. However, if the size of a message is limited to $\tilde{O}(1)$, the number of messages required is $O(m)$ [Cid88, LMT87, Tsi02]. Note that some of these algorithms also works on stricter models for distributed computation. The details on these results can be found in [Tsi02].

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

1.4.3 Empirical analysis of DFS

Most dynamic graphs in real world are dominated by insertion updates [Kun16, LK14, DH14]. Despite its significance, no empirical study have been performed for maintaining a DFS tree even incrementally. For the sake of empirical analysis, we can also consider the static algorithms to be used as incremental algorithms. A short summary of the current-state-of-the-art of incremental algorithms for DFS tree is as follows. An obvious incremental algorithm is to recompute the whole DFS tree in $O(m+n)$ time from scratch after every edge insertion. Let us call it SDFS henceforth. It was shown by Kapidakis [Kap90] that a DFS tree can be computed in $O(n \log n)$ time for a random graph [ER60, Bol84] if we terminate the traversal as soon as all vertices are visited. Let us call this variant as SDFS-Int. Notice that both these algorithms recompute the DFS tree from scratch after every edge insertion. Moving to the algorithms that avoid this recomputation from scratch, we only have the incremental DFS algorithm for DAGs, say FDFS, by Franciosa et al. [FGN97] as described above.

[12] $G(n, p)$ denotes a random graph where every edge of the graph exists independently with probability p.

1.5 Our results

We successfully address the problem of dynamic DFS from several directions: ranging from theoretical to experimental, sequential to parallel/distributed/semi-streaming, and near optimal amortized to significant worst case guarantees. Our results can be described as follows.

1.5.1 Incremental DFS for undirected graph

Maintaining a DFS tree incrementally for an undirected graph (or general directed graph) was stated as an open problem by Franciosa et al. [FGN97]. We present the first incremental algorithm for maintaining a DFS tree (or DFS forest if the graph is not connected) in an undirected graph.

In order to handle insertion of cross edges efficiently, we use two principles. The first principle, called *monotonic fall*, ensures that the depth of each vertex never decreases during the algorithm. The second principle, *minimal restructuring*, ensures that upon insertion of a cross edge, we perform minimal changes to current DFS tree to build the DFS tree of the updated graph. While none of these two principles in isolation is effective, it is their novel combination that leads to an efficient incremental algorithm for a DFS tree. Using these principles, we first present a simple algorithm that achieves total update time of $O(n^{3/2}\sqrt{m})$. We then show how a simple yet significant change to the above algorithm gives a an algorithm which takes a total of $O(n^2)$ time to process any arbitrary online sequence of edges. Observe that the amortized update time per edge insertion for or second algorithm is $\Omega(n^2/m)$, which is $O(1)$ for dense graphs, i.e., $m = \Theta(n^2)$.

A standard way of storing any rooted tree is by keeping a parent pointer for each vertex in the tree. We call this representation an *explicit* representation of a tree. Our algorithm maintains a DFS tree explicitly at each stage. Baswana and Choudhary [BC15] recently established a worst case lower bound of $\Omega(mn)$ for maintaining the ordered DFS tree explicitly under insertion (or deletion) of edges. In the light of this lower bound, our algorithm implies that maintaining a DFS tree explicitly in the incremental environment is provably faster than maintaining an ordered DFS tree for dense graphs.

We also show the existence of a sequence of $\Theta(n)$ edge insertions such that any incremental algorithm that obeys the principle of *monotonic fall* must require $\Omega(n^2)$ time for maintaining a DFS tree explicitly. Therefore, the $O(n^2)$ time complexity of our algorithm is indeed tight even for sparse graphs.

Furthermore, our algorithm uses only $O(m + n)$ extra space. Excluding the standard data structures for maintaining ancestors in a rooted tree [AH00, CH05], our algorithm employs very simple data structures. These salient features make our algorithm an ideal candidate for practical applications as well.

1.5.2 Dynamic DFS with worst case bounds

The existing algorithms described in Section 1.4.1 and our algorithms described above are only designed for partially dynamic environment. Moreover, none of these algorithms achieve a worst case bound of $o(m)$ on the update time. Furthermore, none of these results proves that general DFS is not as hard as ordered DFS in the dynamic environment. This is because the speculations of having to incur a complete recomputation in the worst

case after an update is not disproved by amortized bounds resulting in the perceived $O(m)$ *barrier* for general DFS as well. So the following intriguing questions remained unanswered till date:

- Does there exist any fully dynamic algorithm for maintaining a DFS tree?
- Is it possible to achieve worst case $o(m)$ update time for maintaining a DFS tree in a dynamic environment?

Not only do we answer these open questions affirmatively for undirected graphs, we also use our dynamic algorithm for maintaining a DFS tree to provide efficient solutions for a couple of well studied dynamic graph problems. Moreover, our results also handle vertex updates which are generally considered harder than edge updates. Furthermore, our results finally prove that general DFS is indeed not as hard as ordered DFS in the dynamic setting as was the case in parallel setting. Our results can be described as follows.

We consider a generalized notion of updates wherein an update could be either insertion/deletion of a vertex or insertion/deletion of an edge. For any set U of such updates, let $G+U$ denote the graph obtained after performing the updates U on the graph G. Our main result can be succinctly described in the following theorem.

Theorem 1.1. *An undirected graph can be preprocessed to build a data structure of $O(m \log n)$ size such that for any set U of $k \leq n$ updates, a DFS tree of $G + U$ can be reported in $O(nk \log^4 n)$ time.*

With this result at the core, we easily obtain the following results for dynamic DFS tree in an undirected graph.

1. *Fault Tolerant DFS tree*:

   Given any set of k failed vertices or edges, we can report a DFS tree for the resulting graph in $O(nk \log^4 n)$ time.

2. *Fully Dynamic DFS tree*:

   Given any arbitrary online sequence of vertex or edge updates, we can maintain a DFS tree in $O(\sqrt{mn} \log^{2.5} n)$ worst case time per update.

3. *Incremental DFS tree*:

   Given any arbitrary online sequence of edge insertions, we can maintain a DFS tree in $O(n \log^3 n)$ worst case time per edge insertion.

These are the first $o(m)$ worst case update time algorithms (for sufficiently dense graphs, i.e., $m > n \log^5 n$) to maintain a DFS tree in a dynamic environment. Table 1.1 presents a comparison of our results with the existing results for maintaining a DFS tree in the dynamic setting. Recently, there has been significant work [AW14, HKNS15] on establishing conditional lower bounds on the time complexity of various dynamic graph problems. A simple reduction from [1], based on the Strong Exponential Time Hypothesis (SETH), implies a conditional lower bound of $\Omega(n)$ on the update time of any fully dynamic algorithm for a DFS tree under vertex updates. We also present an unconditional lower bound of $\Omega(n)$ for maintaining a fully dynamic DFS tree explicitly under edge updates.

Graph	Type	Update	Reference
DAG	Incremental	$O(n)$ amortized	Franciosa et al. [FGN97]
DAG	Decremental	$\tilde{O}(n)$ expected amortized	Baswana and Choudhary [BC15]
Undirected	Incremental	$O(n^2/m)$ amortized	**Our Result (Ch. 2)**
Undirected	Incremental	$\tilde{O}(n)$	**Our Result (Ch. 3)**
Undirected	Fault Tolerant *	$\tilde{O}(nk)$	**Our Result (Ch. 3)**
Undirected	Fully Dynamic *	$\tilde{O}(\sqrt{mn})$	**Our Result (Ch. 3)**

Table 1.1: Comparison of different algorithms for maintaining dynamic DFS of a graph. (*) denotes the algorithm also handles vertex updates.

Applications of Fully Dynamic DFS

In the static setting, a DFS tree can be easily used to answer connectivity, 2-edge connectivity and biconnectivity queries. Our fully dynamic DFS algorithm thus seamlessly solves these problems for both vertex and edge updates. Further, our result gives the first deterministic algorithm with $O(1)$ query time and $o(m)$ worst case update time for several well studied variants of these problems in the dynamic setting. These problems include dynamic subgraph connectivity [CPR08, Dua10, EGIN97, Fre85, HdLT01, KKM13] and vertex update versions of dynamic biconnectivity [Hen00, Hen95, HdLT01] and dynamic 2-edge connectivity [HdLT01, EGIN97, Fre85]. The existing results offer different trade-offs between the update time and the query time, and differ on the types (amortized or worst case) of update time and the types (deterministic or randomized) of query time. Our algorithm, in particular, improves the deterministic worst case bounds for these problems, thus demonstrating the relevance of DFS trees in solving dynamic graph problems.

1.5.3 Other Models of Computation

As described earlier in Section 1.4.2, in order to maintain a DFS tree in dynamic setting, each update requires $\tilde{O}(\sqrt{n})$ time on a CRCW PRAM in deterministic parallel setting, $O(n)$ passes in the semi-streaming setting and $O(n)$ rounds in the distributed setting, which is very inefficient. Hence, exploring the dynamic maintenance of a DFS tree in parallel, semi-streaming and distributed environments seems to be a long neglected problem of practical significance.

Again, 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. We presented the first parallel, semi-streaming and distributed algorithms for maintaining a DFS tree in the dynamic setting. Each of these algorithms are nearly optimal (up to $poly \log n$ factors) as described below.

Parallel Algorithms

In the parallel setting, our main result can be succinctly described as follows.

Theorem 1.2. *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 any arbitrary online sequence of vertex or edge 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.

Table 1.2 illustrates our results with respect to the existing result in the right perspective. 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 (up to *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 a fully dynamic DFS tree for an undirected graph is in *NC* (which is still an open problem for DFS tree in the static setting).

Type	**Update**	**Processors**	**Reference**
Fully Dynamic	$\tilde{O}(\sqrt{n})$	n^3	Goldberg et al. [GPV93]
Fully Dynamic	$\tilde{O}(1)$	m	**Our Result (Ch. 4)**
Fault Tolerant	$\tilde{O}(k \log^{2k} n)$	n	**Our Result (Ch. 4)**

Table 1.2: Deterministic Parallel Algorithms for Dynamic DFS

Semi-streaming algorithm

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

Theorem 1.3. *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.*

Table 1.3 illustrates our result with respect to the existing result in the right perspective. Our semi-streaming algorithm clearly takes optimal number of passes (up to *poly* $\log n$ factors) for maintaining a DFS tree.

Type	Passes	Space	Reference
Fully Dynamic	n	$O(n)$	Trivial
Fully Dynamic	$\tilde{O}(1)$	$O(n)$	**Our Result (Ch. 4)**

Table 1.3: Semi-streaming Algorithms for Dynamic DFS

Distributed algorithm

Our parallel fully dynamic DFS algorithm can also be adapted to the distributed environment as follows.

Theorem 1.4. *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.*

Table 1.4 illustrates our result with respect to the existing results in the right perspective. Our distributed algorithm that works in a restricted $\mathcal{CONGEST}(B)$ 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 [13]. This essentially improves the state of the art for the classes of graphs with $o(n)$ diameter.

Type	Rounds	Msg. Size	Messages	Reference
Fully Dynamic	$O(n)$	1	$O(m)$	[Cid88, LMT87, Tsi02]
Fully Dynamic	$O(n)$	n	$O(n)$	[KIS90, MH96, SI89]
Fully Dynamic	$\tilde{O}(D)$	n/D	$\tilde{O}(m + nD)$	**Our Result (Ch. 4)**

Table 1.4: Distributed Algorithms for Dynamic DFS

1.5.4 Empirical analysis of Incremental DFS

We focus on only incremental DFS algorithms as most dynamic graphs in the real world are dominated by insertion updates [Kun16, LK14, DH14]. Moreover, in every other dynamic setting, only a single dynamic DFS algorithm is known making a comparative study impractical. Despite having several algorithms for incremental DFS, not much is known about their empirical performance. Our incremental algorithms having a total update time of $O(n^{3/2}\sqrt{m})$ and $O(n^2)$ will henceforth be referred as ADFS1 and ADFS2 respectively. Whereas, our incremental algorithm with a worst case guarantee of $O(n \log^3 n)$ per update shall be referred as WDFS. Table 1.5 compares the existing algorithms (see Section 1.4.3) and our proposed algorithms for maintaining DFS trees incrementally. However, till date there is no non-trivial incremental DFS algorithm in general directed graphs.

[13] For an algorithm maintaining the whole DFS tree at each node, our message size is also optimal. This is because an update of size $O(n)$ (vertex insertion with arbitrary set of edges) will have to be propagated throughout the network in the worst case. In $O(D)$ rounds, it can only be propagated using messages of size $\Omega(n/D)$. (see Section 4.10.2 for details).

Graph	Time per update	Total time	Reference
Any	$O(m)$	$O(m^2)$	SDFS [Tar72]
Random	$O(n \log n)$ expected	$O(mn \log n)$ expected	SDFS-Int [Kap90]
DAG	$O(n)$ amortized	$O(mn)$	FDFS [FGN97]
Undirected	$O(n^{3/2}/\sqrt{m})$ amortized	$O(n^{3/2}\sqrt{m})$	ADFS1 (Ch. 2)
Undirected	$O(n^2/m)$ amortized	$O(n^2)$	ADFS2 (Ch. 2)
Undirected	$O(n \log^3 n)$	$O(mn \log^3 n)$	WDFS (Ch. 3)

Table 1.5: Comparison of different algorithms for maintaining incremental DFS of a graph.

We contribute to both experimental analysis and average-case analysis of the algorithms for incremental DFS. Our analysis reveals the following interesting results.

Experimental performance of the existing algorithms

We first evaluated the performance of the existing algorithms on the insertion of a uniformly random sequence of $\binom{n}{2}$ edges. The most surprising revelation of this evaluation was the similar performance of ADFS1 and ADFS2, despite the difference in their worst case bounds (see Table 1.5). Further, even FDFS performed better on random graphs taking just $\Theta(n^2)$ time. This is quite surprising because the worst case bounds of ADFS1 and FDFS are greater than $\Theta(n^2)$ by a factor of $\sqrt{m/n}$ and m/n respectively. We then show the tightness of their analysis of ADFS1 and FDFS by constructing worst case examples for these algorithms. Their superior performance on random graphs motivated us to explore the structure of a DFS tree in a random graph.

Structure of DFS tree in random graphs

A DFS tree of a random graph can be seen as a broomstick: a possibly long path without any branching (stick) followed by a bushy structure (bristles). As the graph becomes denser, we show that the length of the stick would increase significantly and establish the following result.

Theorem 1.5. *For a random graph $G(n, m)$ with $m = 2^i n \log n$, its DFS tree will have a stick of length at least $n - n/2^i$ with probability $1 - O(1/n)$.*

The length of stick evaluated from our experiments matches perfectly with the value given by Theorem 1.5. It follows from the broomstick structure that the insertion of only the edges with both endpoints in the bristles can change the DFS tree. As follows from Theorem 1.5, the size of bristles decreases as the graph becomes denser. With this insight at the core, we are able to establish $\tilde{O}(n^2)$ bound on ADFS1 and FDFS for a uniformly random sequence of $\binom{n}{2}$ edge insertions.

Remark: It was Sibeyn [Sib01] who first suggested viewing a DFS tree as a broomstick while studying the height of a DFS tree in random graph. However, his definition of *stick* allowed a few branches on the stick as well. Note that our added restriction (absence of branches on the stick) is crucial in deriving our results as is evident from the discussion above.

New algorithms for random and real world graphs

We use the insight about the broomstick structure and Theorem 1.5 to design a much simpler incremental DFS algorithm (referred as SDFS2) that works for both undirected graphs and directed graphs. Despite being very simple, it is shown to theoretically match (up to $\tilde{O}(1)$ factors) and experimentally outperform ADFS and FDFS for dense random graphs.

For real graphs both ADFS and FDFS were found to perform much better than other algorithms including SDFS2. With the insights from ADFS/FDFS, we design two simple algorithms for undirected and directed graphs respectively (both referred as SDFS3), which perform much better than SDFS2. In fact, for directed graphs SDFS3 almost matches the performance of FDFS for most real graphs considered, despite being much simpler to implement as compared to FDFS.

Semi-Streaming Algorithms

Interestingly, both SDFS2 and SDFS3 can also be used as single-pass semi-streaming algorithms for computing a DFS tree of a random graph using $O(n \log n)$ space. This immediately also gives a single-pass semi-streaming algorithm using the same bounds for answering strong connectivity queries incrementally. Strong connectivity is shown [BMM14, Jan14] to require a working memory of $\Omega(\epsilon m)$ to answer these queries with probability greater than $(1 + \epsilon)/2$ in general graphs, for any $\epsilon > 0$. Hence, our algorithms not only give a solution for the problem in semi-streaming setting but also establish the difference in the hardness of the problem in semi-streaming model for general and random graphs.

1.6 Organization of the thesis

We now present a brief outline of our thesis report. Our incremental algorithms for maintaining a DFS tree for undirected graphs with amortized guarantees is presented in Chapter 2. Chapter 3 describes our algorithms maintaining a DFS tree for undirected graphs with worst case guarantees in the fault tolerant, incremental and fully dynamic settings. The extension of these ideas to other models of computation giving near optimal parallel, semi-streaming and distributed algorithms is presented in Chapter 4. Finally, in Chapter 5 the empirical analysis of the incremental DFS algorithms is presented. We conclude the thesis report with concluding remarks and possible directions for future research in Chapter 6.

Chapter 2

Incremental DFS in Undirected Graphs

2.1 Introduction

Prior to our work, the only known algorithm for maintaining a DFS tree in a dynamic setting was a 20 year old result by Franciosa et al. [FGN97]. They presented an algorithm to maintain incremental DFS for a directed acyclic graph, and also mentioned maintaining incremental DFS for undirected graphs as an open problem. We present the first incremental algorithm for maintaining a DFS tree (or DFS forest if the graph is not connected) of an undirected graph. Our algorithm takes a total of $O(n^2)$ time to process any arbitrary online sequence of edges. Observe that the amortized update time per edge insertion is $\Omega(n^2/m)$, which is $O(1)$ for dense graph (i.e., $m = \Theta(n^2)$). The following short discussion may help one realize the non-triviality of maintaining such a DFS tree incrementally.

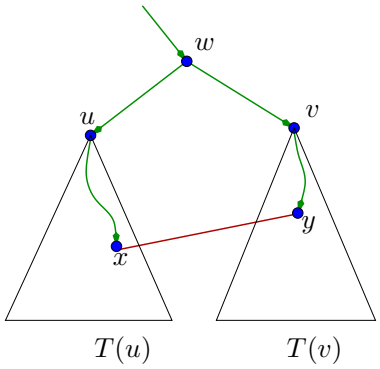

Figure 2.1: Insertion of a cross edge (x, y).

Consider the insertion of an edge (x, y). If (x, y) is a *back edge*, then no change is required in the DFS tree. Otherwise, consider the insertion of a *cross edge* (x, y). See Figure 2.1 for a better visual description. Let w be the lowest common ancestor of x and y in T. Let u and v be its children such that $x \in T(u)$ and $y \in T(v)$. The insertion of (x, y) violates the property of a DFS tree as follows. Let S be the set of visited vertices when the DFS traversal reaches w. Since $T(u)$ and $T(v)$ are two disjoint subtrees hanging from

w, the vertices of $T(u)$ and $T(v)$ belong to disjoint connected components in the subgraph induced by $V \backslash S$. However, the insertion of edge (x, y) connects these components such that the vertices of $T(u) \cup T(v)$ have to hang as a single subtree from w in the DFS tree. This implies that $T(u)$ will have to be rerooted at x and hung from y (or $T(v)$ will have to be rerooted at y and hung from x). This *rerooting* will force restructuring of $T(u)$ because, in order to keep it as a DFS subtree, we need to ensure that each non-tree edge in $T(u)$ is a *back edge*. It is not obvious how to perform this restructuring in an efficient manner.

In order to handle insertion of cross edges efficiently, we use two principles. The first principle, called *monotonic fall*, ensures that the depth of each vertex never decreases during the algorithm. The second principle, called *minimal restructuring*, ensures that while rerooting a subtree upon insertion of a cross edge, we perform minimal changes in the subtree in order to preserve the DFS property. While none of these two principles in isolation is effective, it is their novel combination that leads to an efficient incremental algorithm for maintaining a DFS tree.

A standard way of storing any rooted tree is by keeping a parent pointer for each vertex in the tree. We call this representation an *explicit* representation of a tree. Our algorithm maintains a DFS tree explicitly at each stage. Baswana and Choudhary [BC15] established a worst case lower bound of $\Omega(mn)$ for maintaining the ordered DFS tree explicitly under insertion (or deletion) of edges. In the light of this lower bound, our algorithm implies that maintaining a DFS tree explicitly in the incremental environment is provably faster than maintaining an ordered DFS tree for dense graphs.

We also show the existence of a sequence of $\Theta(n)$ edge insertions such that any incremental algorithm that obeys the principle of *monotonic fall* must require $\Omega(n^2)$ time for maintaining a DFS tree explicitly. Therefore, the $O(n^2)$ time complexity of our algorithm is indeed tight even for sparse graphs.

Furthermore, our algorithm uses only $O(m + n)$ extra space. Excluding the standard data structures for maintaining ancestors in a rooted tree [AH00, CH05], our algorithm employs very simple data structures. These salient features make our algorithm an ideal candidate for practical applications as well.

2.2 Preliminaries

In this chapter we shall use the following notations in addition to the notations described in Appendix A.

- r : Root of the tree T.
- $\textsc{level}(v)$: Level of a vertex v in T such that $\textsc{level}(r) = 0$, and $\textsc{level}(v) = \textsc{level}\big(par(v)\big) + 1$.
- $\textsc{level}(e)$: Level of an edge $e = (x, y)$ in T such that $\textsc{level}(e) = \min\big(\textsc{level}(x), \textsc{level}(y)\big)$.
- $LA(u, k)$: The ancestor of u at level k in tree T.

We explicitly maintain the level of each vertex during the algorithm. Since the tree grows from the root in the downward direction, a vertex u is said to be at higher level than vertex v if $\textsc{level}(u) < \textsc{level}(v)$. Similarly an edge e is said to be higher than edge e' if $\textsc{level}(e) < \textsc{level}(e')$.

We also maintain the following information about the DFS tree T during the algorithm.

- Each vertex v keeps a pointer to $par(v)$. For the root r, $par(r) = r$.
- For each $v \in T$, we keep a list of all its children in tree T. This facilitates the traversal of $T(v)$ from the vertex v.
- Each vertex v keeps a list $\mathcal{B}(v)$ which consists of all the back edges that originate from $T(v)$ and terminate at $par(v)$. This, apparently uncommon and perhaps unintuitive, way of keeping the back edges leads to efficient implementation of the rerooting procedure. $\mathcal{B}(v)$ is maintained as a circular linked list to enable merging of two lists in $O(1)$ time.

Our algorithm uses the following results for the dynamic version of the Lowest Common Ancestor (LCA) and the Level Ancestors (LA) problems.

Theorem 2.1 (Cole and Hariharan 2005 [CH05])**.** *There exists a dynamic data structure for a rooted tree T that uses linear space and can report $LCA(x, y)$ in $O(1)$ time for any two vertices $x, y \in T$. The data structure supports insertion or deletion of any leaf node in $O(1)$ time.*

Theorem 2.2 (Alstrup and Holm 2000 [AH00])**.** *There exists a dynamic data structure for a rooted tree T that uses linear space and can report $LA(u, k)$ in $O(1)$ time for any vertex $u \in T$. The data structure supports insertion of any leaf node in $O(1)$ time.*

The data structure for the Level Ancestor problem, as stated in Theorem 2.2, can be easily extended to handle the deletion of a leaf node in amortized $O(1)$ time using the standard technique of *periodic rebuilding*.

2.3 Overview of the algorithm

Our algorithm is based on two principles. The first principle, called *monotonic fall* of vertices, ensures that the level of a vertex may only fall or remain the same as the edges are inserted. Consider insertion of a cross edge (x, y) as shown in Figure 2.1. In order to ensure monotonic fall, the following strategy is used. If $\textsc{level}(y) \leq \textsc{level}(x)$, then we reroot $T(v)$ at y and hang it through edge (x, y). Otherwise, we reroot $T(u)$ at x and hang it through edge (y, x). This strategy surely leads to a fall in the level of x (or y). However, this rerooting has to be followed by transformation of $T(v)$ into a DFS tree. An obvious, but inefficient, way to do this transformation is to perform a fresh DFS traversal on $T(v)$ from x as done by Franciosa et al. [FGN97] in case of DAG. Moreover, such a traversal may violate the *monotonic fall* of some vertices in $T(v)$. We are able to avoid this costly step using our second principle called *minimal restructuring*. Following this principle, only a path of the subtree $T(v)$ is reversed and as a result, major portion of the original DFS tree remains intact. In fact, this principle also facilitates monotonic fall of all vertices of $T(v)$. The rerooting procedure based on this principle is described and analyzed in the following section.

Our algorithm updates DFS tree upon insertion of any cross edge as follows. Firstly, we carry out rerooting based on the two principles mentioned above. As a result, many back edges now potentially become cross edges. All these edges are collected in a pool,

virtually (re)-inserted back into the graph one by one, and processed as *fresh* insertions. This simple iterative algorithm, when analyzed in a straightforward manner, has a time complexity $O(mn)$. However, using a more careful analysis, it can be shown that its time complexity is $O(n^{3/2}m^{1/2})$, which is strictly sub-cubic. We also present a worst case example proving the tightness of this analysis. In order to improve the time complexity further, we process the pool of cross edges in a more structured manner. In particular, we process the highest cross edge first. This leads to our final algorithm that achieves $O(n^2)$ time complexity for any arbitrary sequence of edge insertions. Subsequently, we also prove that this is the best possible time complexity even for sparse graphs that can be achieved by any algorithm that is based on *monotonic fall* . To establish this fact, we present a sequence of $\Theta(n)$ edge insertions for which such that any algorithm abiding *monotonic fall*, that maintains a DFS tree *explicitly*, would require $\Omega(n^2)$ total time.

2.4 Rerooting a Subtree

Consider insertion of an edge (x, y) which happens to be a cross edge with respect to the DFS tree T. Let w be LCA of x and y, and let u and v be the two children of w such that $x \in T(u)$ and $y \in T(v)$. Let $\text{LEVEL}(y) \leq \text{LEVEL}(x)$. See Figure 2.2 for a visual description. As discussed earlier, updating the DFS tree upon insertion of the cross edge (x, y) entails rerooting of subtree $T(v)$ at y and hanging it from x. We now describe an efficient rerooting procedure for this task based on the principle of *minimal restructuring*.

The underlying idea of *minimal restructuring* is to preserve the current tree structure as much as possible. Consider the path $path(y, v) = \langle z_1(= y), z_2, \ldots, z_k(= v)\rangle$. This path appears from v to y in the rooted tree T. Our rerooting procedure reverses this path in T, such that it now starts at y and terminates at v (see Figure 2.2). In order to see how this reversal affects the DFS structure, let us carefully examine $T(v)$.

The subtree $T(v)$ can be visualized as a collection of disjoint trees hanging from $path(v, y)$ as follows. Let T_1 denote the subtree $T(y)$ and let T_2 denote the subtree $T(z_2)\backslash T(z_1)$. In general, T_i denotes the subtree $T(z_i)\backslash T(z_{i-1})$. Upon reversing $path(v, y)$, notice that each subtree remains intact but their ordering gets reversed. Further, the level of each vertex in every subtree T_i surely falls (see Figure 2.2). Let us find the consequence of reversing $path(v, y)$ on all the back edges with at least one endpoint in $T(v)$. Observe that the back edges which originate as well as terminate within the same T_i continue to remain as back edges since tree T_i remains intact. Likewise, any back edge from these subtrees which terminates at any ancestor of v also continues to remain a back edge. However, the back edges originating in $T(v)$ and terminating on w $\big($i.e., $LCA(x, y)\big)$, which were earlier stored in $\mathcal{B}(v)$, will now have to be stored in $\mathcal{B}(u)$. Recall that $\mathcal{B}(v)$ contains the back edges that originate from $T(v)$ and terminate at $par(v)$. Also, notice that the tree edge (w, v) now becomes a back edge and has to be added to $\mathcal{B}(u)$. The remaining back edges are only those which originate from some T_i and terminate at some $z_j, j > i$. All these edges are present in $\mathcal{B}(z_{j-1})$. Some of these back edges may become cross edges due to the reversal of $path(v, y)$ (see Figure 2.2). Their presence violates the DFS property of the new tree. We simply collect these edges in a set $\mathcal{E}_R$ and remove them temporarily from the graph. In summary, our rerooting algorithm just does the following: It traverses $path(v, y)$ from y to v, collects $\mathcal{B}(z_j)$ for each $1 \leq j < k$ in $\mathcal{E}_R$, and reverses $path(v, y)$. The pseudocode of the rerooting process is described in Procedure Reroot. The following

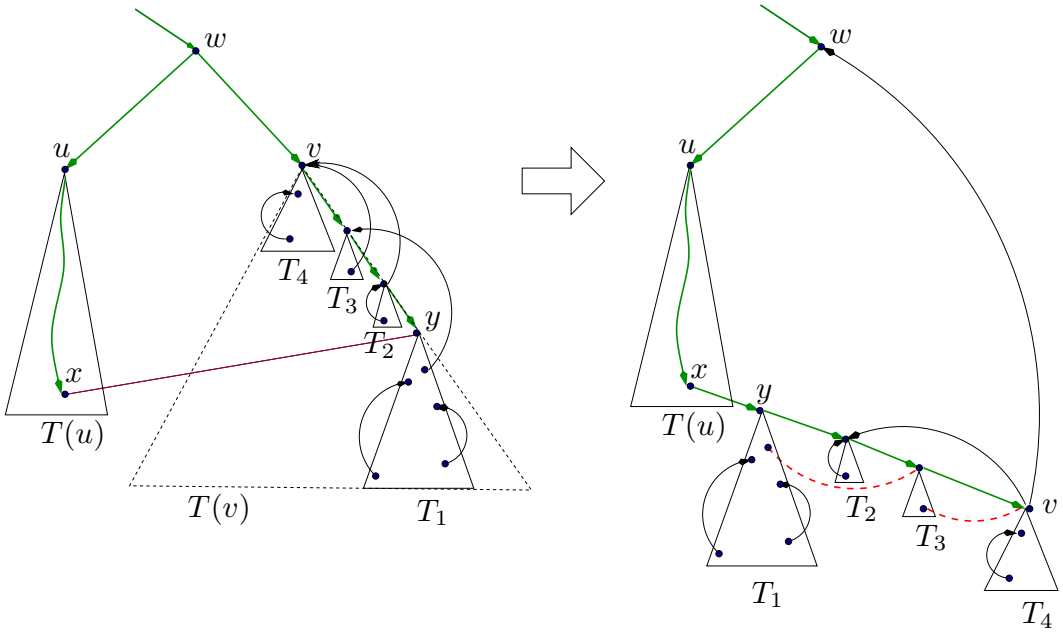

Figure 2.2: Rerooting the tree $T(v)$ at y and hanging it from x. Notice that some back edges may become cross edges (shown dotted in red) due to this rerooting.

lemma follows from the above discussion.

Lemma 2.4.1. *Tree T at the end of Procedure Reroot is a DFS tree for the graph $(V, E \backslash \mathcal{E}_R)$.*

We introduce some terminology to facilitate compact and clean reference to the entities of the rerooting procedure. The lower and higher end vertices x and y of the inserted edge (x, y) are called *prime* and *conjugate* vertices respectively. Notice that the restructured subtree now hangs from the *prime* vertex. We define *prime* path as the path from the prime vertex x to u and *conjugate* path as the path from conjugate vertex y to v, where u and v are children of $LCA(x, y)$ s.t. $x \in T(u)$ and $y \in T(v)$.

Each vertex of the subtree $T(v)$ suffers a fall in its level due to Procedure Reroot. We shall now calculate this fall exactly. Let $\Delta = \text{LEVEL}(x) - \text{LEVEL}(y)$. As a result of the rerooting, y has become child of x. Hence it has suffered a fall in its level by $\Delta + 1$. Since $T_1 = T(y)$ and T_1 remains intact, so each vertex of T_1 also suffers the same fall (of $\Delta + 1$ levels) as y. Consider a vertex z_i which is the root of T_i for some $i > 1$. This vertex was earlier at level $i - 1$ higher than $y(= z_1)$ and now lies at $i - 1$ level below y. Hence overall level of z_i (and hence that of every vertex of T_i) has fallen by $\Delta + 2i - 1$. This leads us to the following lemma.

Lemma 2.4.2. *Let Δ be the difference in the levels of prime and conjugate vertices before rerooting. After rerooting, the i^{th} vertex on the conjugate path (starting from the conjugate vertex) falls by $\Delta + 2i - 1$ levels.*

Let us analyze the time complexity of Procedure Reroot. It first adds $\mathcal{B}(v)$ to $\mathcal{B}(u)$; this step takes $O(1)$ time since we are merging two lists. It then also adds the edge (w, v)

Procedure Reroot(u,v,x,y): reroots subtree $T(v)$ at vertex y and hangs it through edge (x, y). It also updates the data structure $\mathcal{B}$ for u and every vertex on $path(y, v)$, where u is the sibling of v s.t. $x \in T(u)$. It returns the set of back edges $\mathcal{E}_R$ which could potentially be cross edges after rerooting.

```
1  B(u) ← B(u) ∪ B(v) ∪ {(par(v), v)};
2  E_R ← φ;
3  z ← y ;                                  /* z is the first vertex of path(y, v) */
4  p ← x;
5  while z ≠ par(v) do
6  |  if z ≠ v then E_R ← E_R ∪ B(z);
7  |  B(z) ← φ;
8  |  next ← par(z);
9  |  par(z) ← p ;                          /* updating the parent of z */
10 |  p ← z;
11 |  z ← next ;                           /* z is now the next vertex on path(y, v) */
12 end
13 Return E_R
```

to $\mathcal{B}(u)$. Thereafter, the procedure traverses (and reverses) the conjugate path $path(y, v)$, and collects the edges $\mathcal{B}(z)$ for each $z \in path(y, v) \backslash \{v\}$ in $\mathcal{E}_R$. Hence, we can state the following lemma.

Lemma 2.4.3. *The time complexity of Procedure Reroot is $O(k + |\mathcal{E}_R|)$, where k is the length of the conjugate path and $\mathcal{E}_R$ is the set of edges returned by the procedure.*

It follows from the rerooting procedure that any back edge getting converted to a cross edge is surely collected in $\mathcal{E}_R$. However, not all the edges collected in $\mathcal{E}_R$ necessarily become cross edges after Procedure Reroot. In order to understand this subtle point, observe that $\mathcal{E}_R$ contains all those edges which originate from some vertex in T_i and terminate at some $z_j, i < j < k$. Consider any such edge (a, z_j), $a \in T_i$. If $a \neq z_i$ (root of T_i), then surely (a, z_j) has become a cross edge after reversal of $path(v, y)$. But if $a = z_i$, then it still remains a back edge. So we can state the following lemma which will be crucial in our final algorithm.

Lemma 2.4.4. *If an edge collected in $\mathcal{E}_R$ is a back edge with respect to the modified DFS tree, then both its endpoints must belong to the conjugate path.*

2.5 Algorithm for Incremental DFS

We now describe our algorithm for incremental maintenance of a DFS tree. Consider the insertion of an edge (t, z). In order to update the DFS tree, our algorithm maintains a set $\mathcal{E}$ of edges which is initialized as $\{(t, z)\}$. The algorithm then processes the set $\mathcal{E}$ iteratively as follows. In each iteration, an edge $\big($say $(x, y)\big)$ is extracted from $\mathcal{E}$ using Procedure Choose. If the edge is a back edge, the edge is inserted in the set of back edges $\mathcal{B}$ accordingly and no processing is required. If (x, y) is a cross edge, it is processed as follows. Let w be LCA of x and y, and let v be the child of w such that y is present in subtree

$T(v)$. Without loss of generality, let $\text{LEVEL}(x) \geq \text{LEVEL}(y)$. Procedure Reroot$(u, v, x, y)$ is invoked which reroots subtree $T(v)$ at y and returns a set of edges collected during the procedure. All these edges are extracted from E and added to $\mathcal{E}$. This completes one iteration of the algorithm. The algorithm terminates when $\mathcal{E}$ becomes empty (see Algorithm 1 for pseudocode). The correctness of the algorithm follows directly from the following invariant which is maintained throughout the algorithm:

Invariant: T is DFS tree for the subgraph $(V, E \backslash \mathcal{E})$.

Algorithm 1: Processing insertion of an edge (t, z)

```
1  E ← {(t, z)} ;                          /* E is a set of edges to be inserted.   */
2  while E ≠ φ do
3  |  (x, y) ← Choose(E) ;                          /* Let LEVEL(x) ≥ LEVEL(y).   */
4  |  w ← LCA(x, y);
5  |  u ← LA(x, LEVEL(w) + 1) ;     /* u is the child of w s.t.  x ∈ T(u)  */
6  |  v ← LA(y, LEVEL(w) + 1) ;     /* v is the child of w s.t.  y ∈ T(v)  */
7  |  if w ≠ y then                              /* (x, y) is a cross edge.   */
8  |  |  E ← E ∪ Reroot(u, v, x, y);
9  |  end
10 end
```

Procedure Choose($\mathcal{E}$): Chooses and returns an edge from $\mathcal{E}$.

Remove an arbitrary edge (x, y) from $\mathcal{E}$.
Return (x, y).

Furthermore, the LCA and LA data structures introduced in Theorem 2.1 and Theorem 2.2 have to be updated after every iteration of the algorithm. Also, since we maintain the level of each vertex explicitly, $\text{LEVEL}(z)$ has to be updated for each $z \in T(v)$. Following section describes how to perform these updates efficiently.

2.5.1 Maintaining LCA and LA dynamically

Algorithm 1 requires us to answer LCA and LA queries efficiently. The data structures introduced in Theorem 2.1 and Theorem 2.2 maintain LCA and LA in the dynamic setting and answer each query in $O(1)$ time. These data structures allow only leaf updates in the underlying tree (in our case T). However, Procedure Reroot inserts an edge (x, y) and deletes an edge (w, v) in T that leads to rerooting the subtree $T(v)$ at the vertex y (see Figure 2.2). These edges may not be the leaf edges hence these operations are not directly supported by these data structures.

To perform these updates all the edges in $T(v)$ are deleted by iteratively deleting the leaves of $T(v)$. Now, the subtree $T(y)$ is rebuilt at y iteratively by a series of leaf insertions. We know that each vertex in $T(v)$ falls by at least one level during the rerooting event. Note that each falling vertex leads to exactly one leaf insertion and exactly one leaf deletion during this update process, each taking $O(1)$ amortized time. Also, for every falling vertex $z \in T(v)$ we can update $\text{LEVEL}(z)$ in $O(1)$ time. Since each vertex can fall at most n times during the algorithm (ensured by *monotonic fall*), the overall time taken to maintain these data structures (for maintaining LEVEL, LCA and LA) throughout the algorithm is $O(n^2)$.

2.5.2 Analysis

The computation cost of collecting and processing each edge $e \in \mathcal{E}$ can be associated with the rerooting event in which it was collected. Thus, the time spent by the incremental algorithm in processing any sequence of edge insertions is of the order of the time spent in all the rerooting calls invoked and the time spent in maintaining the data structures LEVEL, LCA and LA. Furthermore, using Lemma 2.4.3 we know that the time spent in Procedure Reroot is of the order of the number of edges in $\mathcal{E}_R$ that were collected during the rerooting event and the length of the corresponding conjugate path. However, the cost of traversing the conjugate path can be associated with the fall of vertices on the conjugate path. Therefore, in order to calculate the time complexity of the algorithm, it suffices to count all the edges collected during various rerooting calls and the cost associated with the fall of vertices. Note that this count of collected edges can be much larger than $O(m)$ because an edge can appear multiple times in $\mathcal{E}$ during the algorithm. As described earlier in Section 2.5.1, the overall cost associated with the fall of vertices is $O(n^2)$. It follows from Lemma 2.4.2 that whenever an edge is collected during a rerooting call, the level of at least one of its endpoints falls. Since level of any vertex can fall only up to n, it follows that the computation associated with a single edge during the algorithm is of the order of n. Hence, the $O(mn)$ time complexity of the algorithm is immediate. However, using a more careful insight into the rerooting procedure, we shall now show that the time complexity is much better.

Consider any execution of the rerooting procedure. Let $\langle (y =) z_1, z_2, \ldots, z_k (= v) \rangle$ be the path that gets reversed during the rerooting process (see Figure 2.2). The procedure collects the edges $\mathcal{B}(z_i)$ for each $i < k$. We shall now *charge* each edge collected to the fall of one of its endpoints. Let τ be a parameter whose value will be fixed later. Consider any edge (a, z_i) that is collected during the rerooting process. Note that level of each of a and z_i has fallen due to the rerooting. If $i \leq \tau$, we charge this edge to the fall of vertex a. In this way, there will be at most τ edges that get charged to the fall of a. If $i > \tau$, we charge this edge to the fall of z_i. It follows from Lemma 2.4.2 that z_i falls by at least $2i - 1 > \tau$ levels in this case.

Consider any vertex v in the graph. During a single rerooting event if v falls by less than τ levels, we call it a *short* fall; otherwise we call it a *long* fall for v. It follows that v can be charged for $O(\tau)$ edges in each of its short falls. The number of short falls for v is $O(n)$, so overall cost charged to v due to all its short falls is $O(n\tau)$. On the other hand, v can be charged for $O\big(\deg(v)\big)$ edges in each of its long falls. The number of long falls of v during the entire algorithm is less than n/τ. So the overall cost charged to all the long falls of v will be $O(\deg(v) \cdot n/\tau)$. Hence for all vertices, the total computation charged will be $O(n^2\tau + mn/\tau)$. Fixing $\tau = \sqrt{m/n}$, we can conclude that the overall computation performed during processing of any sequence of m edge insertions by the algorithm is $O(n^{3/2}m^{1/2})$.

Theorem 2.3. *For an undirected graph G on n vertices, a DFS Tree can be maintained incrementally in $O(n^{3/2}m^{1/2})$ total time for any arbitrary sequence of edge insertions.*

It follows from Theorem 2.3 that even for any sequence of $\Theta(n^2)$ edge insertions, the total update time in maintaining DFS tree is $O(n^{2.5})$ which is strictly sub-cubic. In the following section we describe an example to demonstrate the tightness of our analysis.

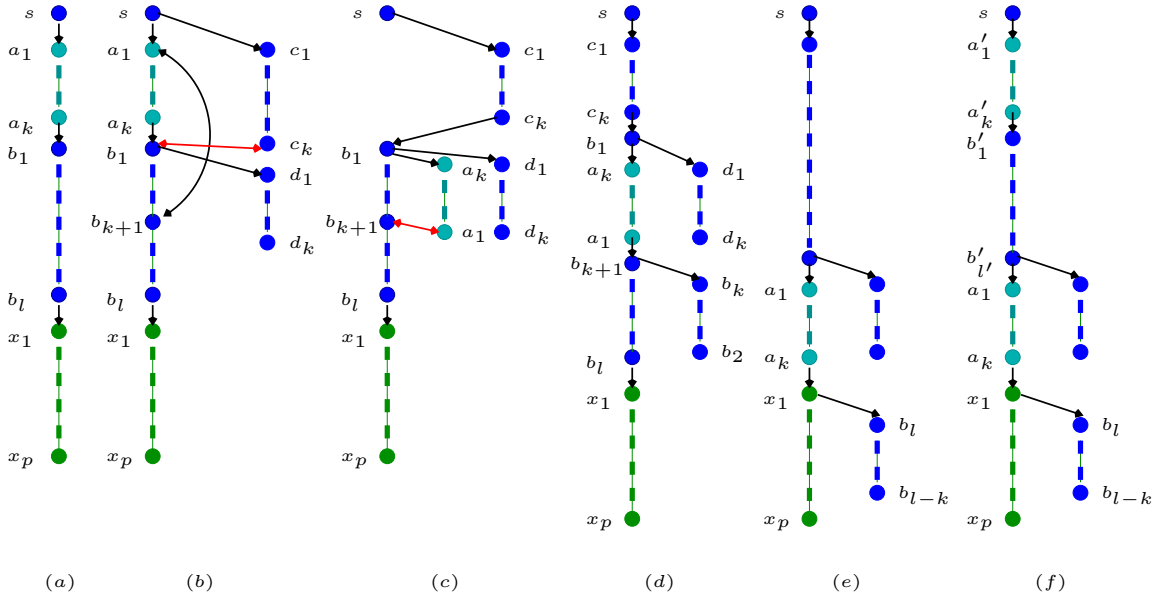

Figure 2.3: Example to demonstrate the tightness of the analysis of Algorithm 1. (a) Beginning of a phase with vertex sets A, B and X. (b) Phase begins with addition of two vertex sets C and D. The first stage begins with the addition of the back edge (a_1, b_{k+1}) and the cross edge (b_1, c_k). (c) The rerooted subtree with the edges in $A \times X$ and (b_{k+1}, a_1) as cross edges. (d) Final DFS tree after the first stage. (e) Final DFS tree after first phase. (f) New vertex sets A', B' and X for the next phase.

2.5.3 Tightness of analysis

We now describe a sequence of $\Theta(m)$ edge insertions for which Algorithm 1 takes $\Theta(n^{3/2}m^{1/2})$ time. Consider a graph $G = (V, E)$ where the set of vertices V is divided into two sets V' and I, each of size $\Theta(n)$. The vertices in V' are connected in the form of a chain (see Figure 2.3 (a)) and the vertices in I are isolated vertices. Thus, it is sufficient to describe only the maintenance of DFS tree for the vertices in set V', as the vertices in I will exist as isolated vertices connected to the dummy vertex s in the DFS tree (recall that s is the root of the DFS tree).

We divide the sequence of edge insertions into n_p *phases*, where each phase is further divided into n_s *stages*. At the beginning of each *phase*, we identify three vertex sets from the set V', namely $A = \{a_1, ..., a_k\}$, $B = \{b_1, ..., b_l\}$ and $X = \{x_1, ..., x_p\}$, where $k, l, p \leq n$ are integers whose values will be fixed later. The value of l is p in the first phase and decreases by k in each subsequent phase. Figure 2.3 (a) shows the DFS tree of the initial graph. We add pk edges of the set $A \times X$ to the graph. Clearly, the DFS tree does not change since all the inserted edges are back edges. Further, we extract two sets of vertices $C = \{c_1, ..., c_k\}$ and $D = \{d_1, ..., d_k\}$ from I and connect them in the form of a chain as shown in Figure 2.3 (b).

Now, the *first stage* of each phase starts with the addition of the back edge (a_1, b_{k+1}) followed by the cross edge (b_1, c_k). As a result Algorithm 1 will reroot the tree $T(a_1)$ as shown in the Figure 2.3 (c) (extra $O(1)$ vertices from I are added to C to ensure the fall of $T(a_1)$ instead of $T(c_1)$). This rerooting makes (b_{k+1}, a_1) a cross edge. Moreover, all pk

edges of set $A \times X$ also become cross edges. Algorithm 1 will collect all these edges and add them to $\mathcal{E}$ in $\Omega(pk)$ time. Since the algorithm can process the edges in $\mathcal{E}$ arbitrarily, it can choose to process (b_{k+1}, a_1) first. It will result in the final DFS tree as shown in Figure 2.3 (d), converting all the cross edges in $A \times X$ to back edges and bringing an end to the first stage. Note the similarity between Figure 2.3 (b) and Figure 2.3 (d): the set C is replaced by the set D, and the set D is replaced by the top k vertices of B. Hence, in the next stage the same rerooting event can be repeated by adding the edges (a_k, b_{2k+1}) and (b_{k+1}, d_k), and so on for the subsequent stages. Now, in every stage the length of B decreases by k vertices. Hence, the stage can be repeated $n_s = O(l/k)$ times in the phase, till A reaches next to X as shown in Figure 2.3 (e). This completes the first phase. Now, in the next phase, first k vertices of the new tree forms the set A' followed by the vertices of B' leading up to the previous A as shown in Figure 2.3 (f). Since the initial size of I is $\Theta(n)$ and the initial size of B is $l < n$, this process can continue for $n_p = O(l/k)$ phases. Hence, each phase reduces the size of I as well as B by k vertices.

Hence, at the beginning of each *phase*, we extract $2k$ isolated vertices from I and add pk edges to the graph. In each *stage*, we extract $O(1)$ vertices from I and add just 2 edges to the graph in such a way that will force our algorithm to process pk edges to update the DFS tree. Thus, the total number of edges added to the graph is $pk \cdot n_p$ and the total time taken by our algorithm is $pk \cdot n_p \cdot n_s$, where $n_p = O(l/k)$ and $n_s = O(l/k)$. Substituting $l = n, p = m/n$ and $k = \sqrt{m/n}$, we have a sequence of m edge insertions for which our algorithm takes $\Theta(n^{3/2}m^{1/2})$. Hence for any $n \leq m \leq \binom{n}{2}$, we get the following theorem.

Theorem 2.4. *For each value of $n \leq m \leq \binom{n}{2}$, there exists a sequence of m edge insertions for which Algorithm 1 requires $\Theta(n^{3/2}m^{1/2})$ time to maintain the DFS tree.*

Remark: The core of this example is the rerooting event occurring in each stage that takes $\Theta(m^{3/2}/n^{3/2})$ time. This event is repeated systematically $n_s \cdot n_p$ times to force the algorithm to take $\Theta(n^{3/2}m^{1/2})$ time. However, this is possible only because our algorithm processes $\mathcal{E}$ arbitrarily: processing the cross edge (b_k, a_1) first amongst all the collected cross edges. Note that, had the algorithm processed any other cross edge first, we would have reached the end of a phase in a single stage. The overall time taken by the algorithm for this example would then be just $\Theta(m)$. Interestingly, with just a more structured way of processing the edges of $\mathcal{E}$, we can even achieve a worst case bound of $O(n^2)$ for our algorithm. We provide this improved algorithm in the following section.

2.6 Achieving $O(n^2)$ update time

The time complexity of Algorithm 1 is governed by the number of edges in $\mathcal{E}$ that are processed during the algorithm. In order to get an improved algorithm, let us examine $\mathcal{E}$ carefully. An edge from $\mathcal{E}$ can be a cross edge or a back edge. Processing of a cross edge always triggers a rerooting event which, in turn, leads to fall of one or more vertices. Hence, the total number of cross edges processed during the algorithm is $O(n^2)$. All the remaining edges processed in $\mathcal{E}$ during the entire algorithm are back edges. There are two sources of these back edges.

Firstly, some edges added to $\mathcal{E}$ by Procedure Reroot are back edges. Let us analyze their count throughout the algorithm. It follows from Lemma 2.4.4 that both endpoints of each such back edge belong to the conjugate path associated with the rerooting event.

Notice that i^{th} vertex on the conjugate path falls by at least $2i-1$ levels (see Lemma 2.4.2). So, if ℓ is the length of the conjugate path, the total fall in the level of all vertices on the conjugate path is more than $\ell(\ell-1)/2$, which is also an upper bound on the number of edges with both endpoints on the conjugate path. Since the total fall in the level of vertices cannot be more than $O(n^2)$, the number of such back edges throughout the algorithm is $O(n^2)$.

Secondly, some edges added to $\mathcal{E}$ by Procedure Reroot are cross edges at the time of their collection, but become back edges before they are processed. This may happen due to rerooting initiated by some other cross edge from $\mathcal{E}$. In order to understand this subtle point, see Figure 2.4. Here e_1, e_2, e_3, e_4 and $e_5 = (x, y)$ are cross edges present in $\mathcal{E}$ at some stage. While we process (x, y), $T(v)$ gets rerooted at y and hangs through the edge (x, y). As a result e_1, e_2, e_3 and e_4 become back edges.

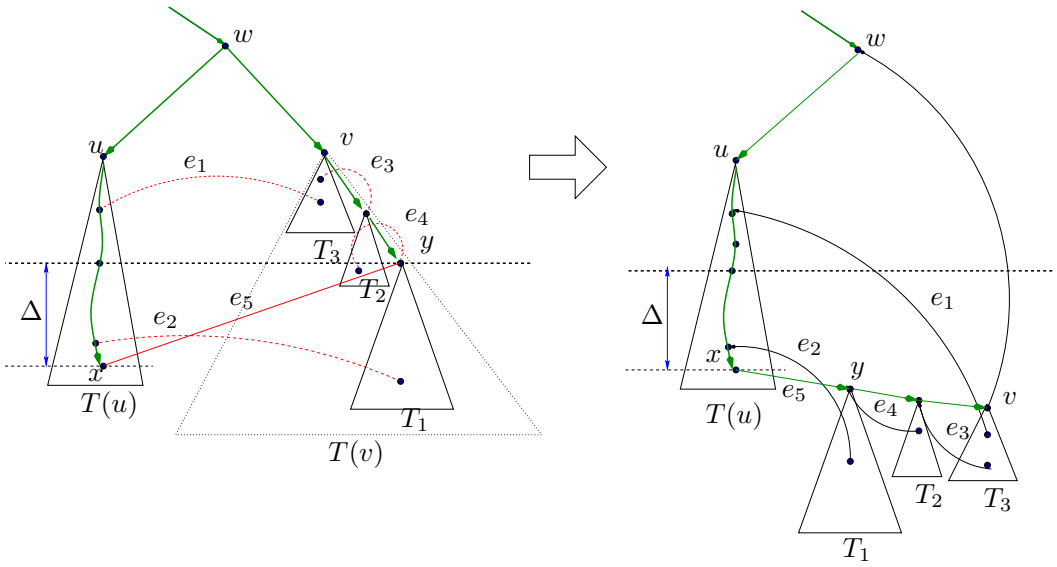

Figure 2.4: Some cross edges in $\mathcal{E}$ become back edges due to the rerooting of $T(v)$.

In order to bound these cross edges getting transformed into back edges during rerooting of $T(v)$, let us carefully examine one such cross edge. Let v_h and v_l be respectively the higher and lower endpoints of the resulting back edge. The edge (v_h, v_l) was earlier a cross edge, so v_h now has a new descendant v_l. Similarly v_l now has a new ancestor v_h. Note that the descendants of only vertices lying on prime and conjugate paths are changed during rerooting. Also the ancestors of only the vertices in $T(v)$ are changed during rerooting. Hence the following lemma holds true.

Lemma 2.6.1. *Cross edges getting converted to back edges as a result of rerooting $T(v)$ are at most $|T(v)|$ times the sum of lengths of prime and conjugate paths.*

Observe that the total sum of the fall of vertices of $T(v)$ during the rerooting event is at least $|T(v)| \cdot (\Delta + 1)$ (see Figure 2.4). However, the sum of the lengths of prime and conjugate paths may be much greater than $\Delta + 1$. Hence, the number of cross edges getting transformed into back edges is not bounded by the total sum of fall of vertices of $T(v)$. This observation, though discouraging, also gives rise to the following insight:

If there were no cross edges with level higher than LEVEL(y), the number of cross edges converted to back edges will be at most the total sum of fall of vertices of $T(v)$. This is because the possible higher endpoints of such edges on the prime or conjugate path will be limited. This insight suggests that processing higher cross edges from $\mathcal{E}$ first will be more advantageous than the lower ones. Our final algorithm is inspired by this idea.

2.6.1 The final algorithm

Our final algorithm is identical to the previous algorithm in Section 2.5 except that instead of invoking Procedure Choose we invoke Procedure ChooseHigh.

Procedure ChooseHigh($\mathcal{E}$): Chooses and returns the highest edge in $\mathcal{E}$.
Remove the highest edge (x, y) from $\mathcal{E}$. Return (x, y).

The algorithm thus processes the edges of set $\mathcal{E}$ in a non-decreasing order of their levels. To achieve this we require a data structure that maintains $\mathcal{E}$ and allows retrieval of edges in the desired order. This data structure should also support insertion of new edges and change in the level of edges due to fall of a vertex efficiently.

This can be achieved by keeping a binary heap on endpoints of the edges in $\mathcal{E}$, where the *key* of each endpoint is its level in T. Thus, each of the two operations mentioned above can be performed in $O(\log n)$ time taking total $O(n^2 \log n)$ time throughout the algorithm. However, this can be improved to total $O(n^2)$ time using a simple data structure that is described in the following section.

2.6.2 Data Structure to maintain set of edges $\mathcal{E}$

Our data structure is an array H, where $H[i]$ ($i \in [1, n]$) stores a list of all the vertices at level i that have at least one of its edges in $\mathcal{E}$. In addition, each vertex v stores a list $\mathcal{L}(v)$ of all its edges that are present in $\mathcal{E}$. It can be observed that upon insertion of an edge in $\mathcal{E}$ or update in the level of a vertex in T, H can be updated in $O(1)$ time.

Consider insertion of any cross edge, say $e = (x, y)$ in the graph. Initially the data structure H is empty. We insert x and y at their corresponding levels in H. We then add e to both $\mathcal{L}(x)$ and $\mathcal{L}(y)$, and also store its pointers in the lists $\mathcal{L}(x)$ and $\mathcal{L}(y)$ to facilitate its deletion from these lists in $O(1)$ time. We now start a scan of H from LEVEL(e) to process the edges in $\mathcal{E}$ in a non-decreasing order of their levels as follows.

Suppose i is the currently scanned index of H. If $H[i]$ is non-empty, we process edges of $\mathcal{L}(v)$ for each $v \in H[i]$ one by one. Processing of an edge may cause a rerooting event that may result in addition of new edges to $\mathcal{E}$. We insert these edges and their corresponding endpoints in H accordingly. In addition, this rerooting may result in the fall of some vertices already present in H. We move these vertices to their new position in H accordingly. After processing an edge, say $e = (u, v)$, e is removed from $\mathcal{L}(u)$ and $\mathcal{L}(v)$. If $\mathcal{L}(u)$ $\big($likewise $\mathcal{L}(v)\big)$ becomes empty, we remove u (likewise v) from H. As a result, all the vertices in $H[i]$ will be removed and $H[i]$ will become empty. Notice that all the edges collected in $\mathcal{E}$ during any rerooting event that occurs while processing $H[i]$ will be at a level lower than i. This is because all these collected edges will be from the subtree now

hanging from the edge that caused the rerooting event. Thus, we just need to continue scanning $H[i]$ for increasing values of i until $\mathcal{E}$ becomes empty.

It follows that the data structure H facilitates processing of the edges in $\mathcal{E}$ in non-decreasing order of their levels. It requires $O(1)$ time for inserting an edge in $\mathcal{E}$ and updating the level of a vertex. The only overhead in maintenance of H is the time required for scanning through the cells of H. This factor becomes significant only when a lot of empty cells have to be scanned in H to find the next highest edge in $\mathcal{E}$. This cost can be charged to the fall of endpoints of edges in $\mathcal{E}$ as follows. Consider any edge $e \in \mathcal{E}$ which is collected during a rerooting event caused by processing some edge, say e_0. The level of edge e may fall several times before it is finally processed and removed from $\mathcal{E}$. The entire cost of scanning through H from $\textsc{level}(e_0)$ to the eventual level of e when it is finally processed is charged as follows. After the rerooting event caused by processing e_0, the difference between $\textsc{level}(e_0)$ and $\textsc{level}(e)$ is always less than the fall of the endpoint of e on the conjugate path. Any subsequent change in $\textsc{level}(e)$ will be charged to the fall of that endpoint of e which led to this change. Hence, the time spent in scanning through H during the entire algorithm is bounded by the total sum of fall of vertices in T, which is $O(n^2)$.

2.6.3 Analysis

In order to establish $O(n^2)$ bound on the running time of our final algorithm, it follows from the preceding discussion that we simply need to show that the number of cross edges getting converted to back edges throughout the algorithm is $O(n^2)$.

Consider any rerooting event initiated by cross edge (x, y) (refer to Figure 2.4). Since there is no edge in $\mathcal{E}$ which is at a higher level than $\textsc{level}(y)$, so it follows from Lemma 2.6.1 that the cross edges getting converted to back edges during the rerooting event will be of one of the following types only.

- The cross edges with one endpoint in $T(v)$ and another endpoint x or any of Δ ancestors of x.
- The cross edges with y as one endpoint and another endpoint anywhere in $T(v)\backslash T(y)$.

Hence the following lemma holds for our final algorithm.

Lemma 2.6.2. *During the rerooting event the number of cross edges converted to back edges are at most* $|T(v)| \cdot (\Delta + 1) + |T(v)\backslash T(y)|$.

According to Lemma 2.4.2, level of each vertex of $T(v)$ falls by at least $\Delta + 1$. So the first term in Lemma 2.6.2 can be clearly associated with the fall of each vertex of $T(v)$. Note that each vertex in $T(v)\backslash T(y)$ becomes a descendant of y and hence falls by at least one extra level (in addition to $\Delta + 1$). This fall by extra one or more levels for vertices of $T(v)\backslash T(y)$ can be associated with the second term mentioned in Lemma 2.6.2. Hence the total number of cross edges getting transformed to back edges during the algorithm is of the order of $O(n^2)$. We can thus state the following theorem.

Theorem 2.5. *For an undirected graph G on n vertices, a DFS Tree can be maintained under insertion of any arbitrary sequence of edges with total update time of $O(n^2)$.*

The $O(n^2)$ bound of our algorithm is quite tight even for sparse graphs. In fact, in the next section we show the following: There exists a sequence of $\Theta(n)$ edge insertions such that every incremental algorithm for maintaining DFS tree *explicitly* (using parent or child pointers) that follows the principle of *monotonic fall* will require $\Omega(n^2)$ time.

2.6.4 Limitations of *monotonic fall*

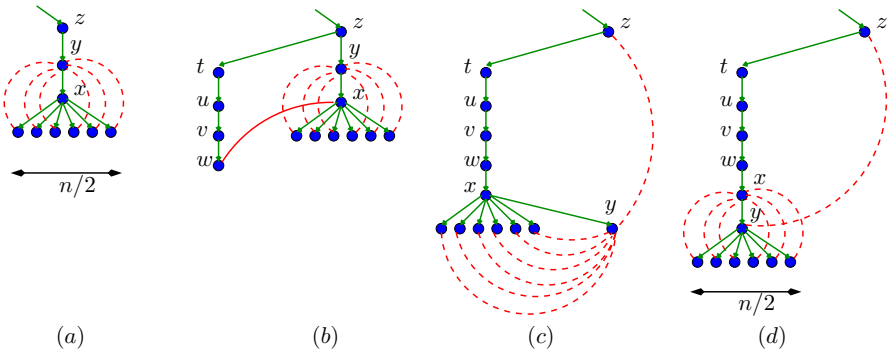

Figure 2.5: Example to show tightness of analysis.

Consider a graph on n vertices with $n/2-3$ isolated vertices in the beginning, and a set P of $n/2$ vertices hanging from a vertex x. Let $par(x) = y$ and $par(y) = z$, and each vertex in P has a back edge to y (see Figure 2.5(a)). Now use 4 isolated vertices t, u, v, w, and connect them using the edges $(z,t),(t,u),(u,v),(v,w)$ followed by insertion of edge (w,x) (see Figure 2.5(b)). If the algorithm follows *monotonic fall*, it must hang y from x (see Figure 2.5(c)) and then eventually hang P from y (see Figure 2.5(d)). This creates a structure similar to the original tree shown in Figure 2.5(a). This process changes $n/2$ tree edges and hence requires at least $\Omega(n)$ time. This step can be repeated $n/8$ times. Hence, overall inserting $\Theta(n)$ edges will require $\Omega(n^2)$ time proving the following theorem.

Theorem 2.6. *Any incremental algorithm that follows the principle of monotonic fall, will require a total update time of $\Omega(n^2)$ for maintaining a DFS Tree explicitly even for sparse graphs.*

2.7 Discussion

We presented a simple and efficient algorithm for maintaining a DFS tree for an undirected graph under insertion of edges. Further, the data structures used by our algorithm (except for that of LCA and LA) are very simple and easy to implement. These salient features make our algorithm an ideal candidate for practical applications.

Even though the amortized update time of our algorithm is optimal for dense graphs, for sparse graphs it is no better than recomputing the DFS tree from scratch after every update. Moreover, we also proved that any algorithm that employs the principle of *monotonic fall* will require $\Omega(n^2)$ time to maintain incremental DFS tree *explicitly* even for sparse graphs. Hence, a totally new approach would be required to improve the result for sparse graphs.

Another, perhaps more significant, research direction would be to develop an algorithm for maintaining DFS tree of a directed graph in any dynamic setting. The current techniques are not applicable in this case, because of the following reasons. *Firstly*, due to lack of flexibility to choose the orientation of the inserted edge, *monotonic fall* cannot be ensured directly as in case of undirected graphs. *Secondly*, simple reversal of a path, which was the basis of *minimal restructuring*, is not possible in case of directed graphs. *Finally*, the simplicity of performing the rerooting of a subtree by recursively performing rerooting of independent subtrees is not evident in case of directed graphs.

Chapter 3

Dynamic DFS with worst case bounds

3.1 Introduction

Prior to this work, very little progress was achieved for maintaining a DFS tree in different dynamic settings. The only known algorithms were by Franciosa et al. [FGN97] and Baswana and Choudhary [BC15] for maintaining incremental and decremental DFS tree respectively in a directed acyclic graph under edge updates. Even our prior work (see Chapter 2) maintained incremental DFS tree for an undirected graph only under edge insertions. However, none of these existing algorithms, though designed for only a partially dynamic environment, achieve a worst case bound of $o(m)$ on the update time. Furthermore, none of these results truly differentiates the hardness of *general DFS* from *ordered DFS* in the dynamic environment (see Section 1.4). This is because efficient amortized bounds does not refute the possibility of a complete recomputation in the worst case after an update, resulting in the conjectured $O(m)$ *barrier* for general DFS as well. Finally, none of the previous results, though significant in theory, was able to demonstrate applications of DFS trees in the dynamic setting, as was the case in the static setting.

In this chapter, we present dynamic DFS algorithms with $o(m)$ worst case update time for undirected graphs in the fault tolerant, incremental and fully dynamic settings. Further, our fault tolerant and fully dynamic algorithms handle both edge and vertex updates, where a vertex can be inserted with an arbitrary set of edges incident on it. Moreover, our results finally prove that general DFS is indeed not as hard as ordered DFS in the dynamic setting. Finally, we use our fully dynamic DFS algorithm to provide efficient solutions for a couple of well studied dynamic graph problems as connectivity, 2-edge connectivity and biconnectivity in the dynamic subgraph model. Our algorithm, in particular, improves the deterministic worst case bounds for these problems, thus demonstrating the relevance of DFS trees in solving dynamic graph problems. We also present conditional lower bounds for dynamic DFS under both edge and vertex updates.

For a given set U of edge/vertex updates, our main idea is use T itself, by preprocessing the graph G using tree T to build a data structure $\mathcal{D}$. To achieve $o(m)$ update time, our algorithm uses $\mathcal{D}$ to create a *reduced* adjacency list for each vertex such that performing DFS traversal using these lists gives a DFS tree of the updated graph $G + U$. These reduced adjacency lists are generated on the fly and necessarily have only $\tilde{O}(n|U|)$ edges.

3.2 Preliminaries

Let U be any given set of updates. We add a dummy vertex r to the given graph in the beginning and connect it to all the vertices. Our algorithm starts with any arbitrary DFS tree T rooted at r in the augmented graph and it maintains a DFS tree rooted at r at each stage. It can be observed easily that each subtree rooted at any child of r is a DFS tree of a connected component of the graph $G + U$. The following notations will be used throughout the chapter.

- $dist_T(x, y)$: The number of edges on the path from x to y in T.
- $N(w)$: The adjacency list of vertex w in the graph $G + U$.
- $L(w)$: The reduced adjacency list of vertex w in the graph $G + U$.

We now state the operations supported by the data structure $\mathcal{D}$ (complete details of $\mathcal{D}$ are in Section 3.4). Let U below refer to a set of updates that consists of vertex and edge deletions only. For any three vertices $w, x, y \in T$, where $path(x, y)$ is an ancestor-descendant path in T the following two queries can be answered using $\mathcal{D}$ in $O(\log^3 n)$ time.

1. $Query(w, x, y)$: among all the edges from w that are incident on $path(x, y)$ in $G+U$, return an edge that is incident nearest to x on $path(x, y)$.

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

We now describe an important property of a DFS traversal that will be crucially used in our algorithm.

Properties of a DFS tree

DFS traversal has the following flexibility : when the traversal reaches a vertex, say v, the next vertex to be traversed can be *any* unvisited neighbor of v. In order to compute a DFS tree for $G + U$ efficiently, our algorithm exploits this flexibility, the original DFS tree T, and the following property of DFS traversal.

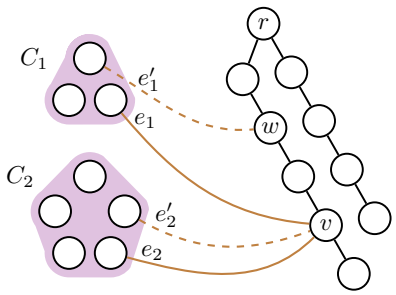

Figure 3.1: Edges e'_1 as well as e'_2 can be ignored during the DFS traversal.

Lemma 3.2.1 (Components Property)**.** *Let T^* be the partially grown DFS tree and v be the vertex currently being visited. Let C be any connected component in the subgraph induced by the unvisited vertices. Suppose two edges e and e' from C are incident respectively on v and some ancestor (not necessarily proper) w of v in T^*. Then it is sufficient to consider only e during the rest of the DFS traversal, i.e., the edge e' need not be scanned. (Refer to Figure 3.1).*

Skipping e' during the DFS traversal, as stated in the components property, is justified because e' will appear as a back edge in the resulting DFS tree. A similar property describing the *inessential* edges of a DFS trees was used by Smith [Smi86] for computing a DFS tree of a planar graph in the parallel setting. In order to highlight the importance of the components property, and to motivate the requirement of data structure $\mathcal{D}$, we first consider a simpler case which deals with reporting a DFS tree after a single update in the graph.

3.3 Handling a single update

Consider the failure of a single edge (b, f) (refer to Figure 3.2 (i)). Exploiting the flexibility of DFS traversal, we can assume a stage in the DFS traversal of $G\backslash\{(b, f)\}$ where the partial DFS tree T^* is $T\backslash T(f)$ and vertex b is currently being visited. Thus, the unvisited graph is a single connected component containing the vertices of $T(f)$. Now, according to the components property we need to process only the lowest edge from $T(f)$ to $path(b, r)$ ((k, b) in Figure 3.2 (ii)). Hence, the DFS traversal enters this component using the edge (k, b) and performs a traversal of the subgraph induced by the vertices of $T(f)$. The resulting DFS tree of this subgraph would now be rooted at k. Rebuilding the DFS tree after the failure of edge (b, f) thus reduces to finding the lowest edge from $T(f)$ to $path(e, r)$, and then rerooting a subtree $T(f)$ of T at the new root k. We now describe how this rerooting can be performed in $\tilde{O}(n)$ time in the following section.

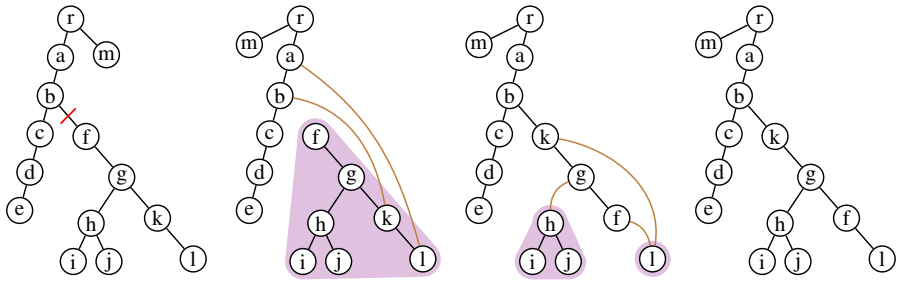

Figure 3.2: (i) Failure of edge (b, f). (ii) Partial DFS tree T^* with unvisited graph $T(f)$, component property allows us to neglect (a, l). (iii) Augmented $path(k, f)$ to T^*, the components property allows us to neglect (l, k). (iv) Final DFS tree of $G\backslash\{(b, f)\}$.

3.3.1 Rerooting a DFS tree

Given a DFS tree T originally rooted at r_0 and a vertex r', the aim is to compute a DFS tree of the graph that is rooted at r'. Note that any subtree $T(x)$ of the DFS tree T is also

a DFS tree of the subgraph induced by the vertices of $T(x)$. Hence, the same procedure can be applied to reroot a subtree $T(x)$ of the DFS tree T. Thus, in general our aim is to reroot $T(r_0)$ at a new root $r' \in T(r_0)$ (see Figure 3.2 (ii), where the subtree $T(f)$ would be rerooted at its new root k).

Procedure Reroot$(T(r_0), r')$: Reroots the subtree $T(r_0)$ of T to be rooted at the vertex $r' \in T(r_0)$.

```
foreach (a, b) on path(r_0, r') do    /* a = par(b) in original tree T(r_0). */
    par(a) ← b;
    foreach child c of b not on path(r_0, r') do
        (u, v) ← Query(T(c), r_0, b) ;  /* where u ∈ path(r_0, r') and v ∈ T(c). */
        if (u, v) is non-null then
            Reroot(T(c), v);
            par(v) ← u;
        end
    end
end
```

Figure 3.3: The recursive algorithm to reroot a DFS tree $T(r_0)$ from the new root r'.

Our algorithm (refer to Procedure Reroot) essentially performs the DFS traversal (exploiting the flexibility of DFS) in such a way that components of the unvisited graph can be easily identified. The components property can then be applied to each such component, processing only $O(n)$ edges to compute the rerooted DFS tree. The DFS traversal first visits the path from r' to the root of tree $T(r_0)$. This reverses $path(r_0, r')$ in the new DFS tree T^* as now r' would be an ancestor of r_0 (see Figure 3.2 (iii)). Now, each subtree hanging from $path(r', r_0)$ in T forms a component of the unvisited graph. This is because the presence of any edge between these subtrees would imply a cross edge in the original DFS tree. Using the components property we know that for each of these subtrees, say T_i, we only need to process the lowest edge from T_i on the new path from r' to r_0 in T^*. Since $path(r', r_0)$ is reversed in T^*, it is equivalent to processing the highest edge e_i from T_i to the $path(r_0, r')$ in T. Recall that this query can be answered by our data structure $\mathcal{D}$ in $O(\log^3 n)$ time (refer to Section 3.2). Now, let v_i be the end vertex of e_i in T_i. The DFS traversal will thus visit the component induced by the vertices of T_i through e_i, and produces its DFS tree that is rooted at v_i. This rerooting can be performed by invoking the rerooting procedure recursively on the subtree T_i with the new root v_i.

We now analyze the total time required by Procedure Reroot to reroot a subtree T' of the DFS tree T. The total time taken by our algorithm is proportional to the number of edges processed by the algorithm. These edges include the *tree edges* that were a part of the original tree T' and the *added edges* that are returned by the data structure $\mathcal{D}$. Clearly, the number of tree edges in T' are $O(|T'|)$. Also, since the added edges eventually become a part of the new DFS tree T^*, they too are bounded by the size of the tree T'. Further, the data structure $\mathcal{D}$ takes $O(\log^3 n)$ time to report each added edge. Hence the total time taken by our algorithm to rebuild T' is $O(|T'| \log^3 n)$ time. Since $\mathcal{D}$ can be built in $O(m \log n)$ time (refer to Theorem 3.3 in the Section 3.4), we have the following theorem.

Theorem 3.1. *An undirected graph can be preprocessed to build a data structure in $O(m \log n)$ time, such that any subtree T' of the DFS tree can be rerooted at any vertex in T', in $O(|T'| \log^3 n)$ time.*

We now formally describe how rebuilding a DFS tree after an update can be reduced to this simple rerooting procedure (see Figure 3.4).

1. **Deletion of an edge (u, v):**
   In case (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 $T(v)$ is then rerooted at its new root v' and hanged from u' using (u', v') in 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 $T(v')$ is then rerooted at its new root v and hanged from u using (u, v) in 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 $T(v_i)$ is then rerooted at its new root v'_i and hanged from u'_i using (u'_i, v'_i) in the final tree T^*.

4. **Insertion of a vertex u:**
   Let $v_1, ..., v_c$ be the neighbors of u in the graph. Make u a child of some 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 $T(v'_i)$ is then rerooted at its new root v_i and hanged from u using (u, v_i) in the final tree T^*.

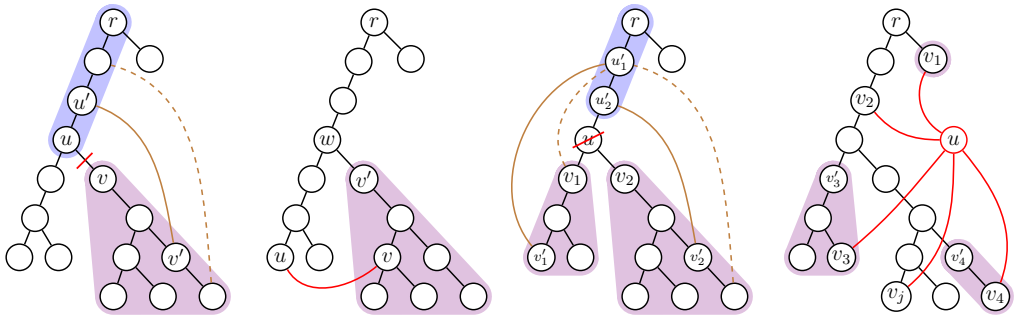

Figure 3.4: Updating the DFS tree after a single update: (i) deletion of an edge, (ii) insertion of an edge, (iii) deletion of a vertex, and (iv) insertion of a vertex. The reduction algorithm reroots the marked subtrees (shown in violet) and hangs it from the inserted edge (in case of insertion) or the lowest edge (in case of deletion) on the marked path (shown in blue) from the marked subtree.

In case of vertex updates, multiple subtrees may be rerooted by the algorithm. Let these subtrees be $T_1, ..., T_c$. Thus, the total time taken by our algorithm is equal to the

time taken to reroot the subtrees $T_1, ..., T_c$. Using Theorem 3.1, we know that a subtree T' can be rerooted in $\tilde{O}(|T'|)$ time. Since these subtrees are disjoint, the total time taken by our algorithm to build the resulting DFS tree is $\tilde{O}(|T_1| + ... + |T_c|) = \tilde{O}(n)$. Thus, we have the following theorem.

Theorem 3.2. *An undirected graph can be preprocessed to build a data structure in $O(m \log n)$ time such that after a single update in the graph, the DFS tree can be reported in $O(n \log^3 n)$ time.*

3.4 Data Structure

The efficiency of our algorithm heavily relies on the data structure $\mathcal{D}$. For any three vertices $w, x, y \in T$, where $path(x, y)$ is an ancestor-descendant path in T, we need to answer the following two kinds of queries.

1. $Query(w, x, y)$: among all the edges from w that are incident on $path(x, y)$ in $G+U$, return an edge that is incident nearest to x on $path(x, y)$.

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

We now describe construction of the data structure $\mathcal{D}$. It employs a combination of two well known techniques, namely, heavy-light decomposition [ST83] and suitable augmentation of a binary tree (segment tree) as follows.

1. Perform a preorder traversal of tree T with the following restriction: Upon visiting a vertex $v \in T$, the child of v that is visited first is the one storing the largest subtree. Let $\mathcal{L}$ be the list of vertices ordered by this traversal.

2. Build a segment tree $\mathcal{T}_\mathcal{B}$ whose leaf nodes from left to right represent the vertices in list $\mathcal{L}$.

3. Augment each node z of $\mathcal{T}_\mathcal{B}$ with a binary search tree $\mathcal{E}(z)$, storing all the edges $(u, v) \in E$ where u is a leaf node in the subtree rooted at z in $\mathcal{T}_\mathcal{B}$. These edges are sorted according to the position of the second endpoint in $\mathcal{L}$.

The construction of $\mathcal{D}$ described above ensures the following properties which are helpful in answering a query $Query(T(w), x, y)$ (see Figure 3.5).

- $T(w)$ is present as an interval of vertices in $\mathcal{L}$ (by step 1). Moreover, this interval can be expressed as a union of $O(\log n)$ disjoint subtrees in $\mathcal{T}_\mathcal{B}$ (by step 2). Let these subtrees be $\mathcal{T}_\mathcal{B}(z_1), \ldots, \mathcal{T}_\mathcal{B}(z_q)$.

- It follows from the heavy-light decomposition used in step 1 that path $path(x, y)$ can be divided into $O(\log n)$ subpaths $path(x_1, y_1), \ldots, path(x_\ell, y_\ell)$ such that each subpath $path(x_i, y_i)$ is an interval in $\mathcal{L}$.

- Let query $Q(z, x, y)$ return the edge on $path(x, y)$ from the vertices in the subtree $\mathcal{T}_\mathcal{B}(z)$, that is closest to vertex x. Then it follows from step 3 that any query $Q(z_j, x_i, y_i)$ can be answered by a single predecessor or successor query on BST $\mathcal{E}(z_j)$ in $O(\log n)$ time.

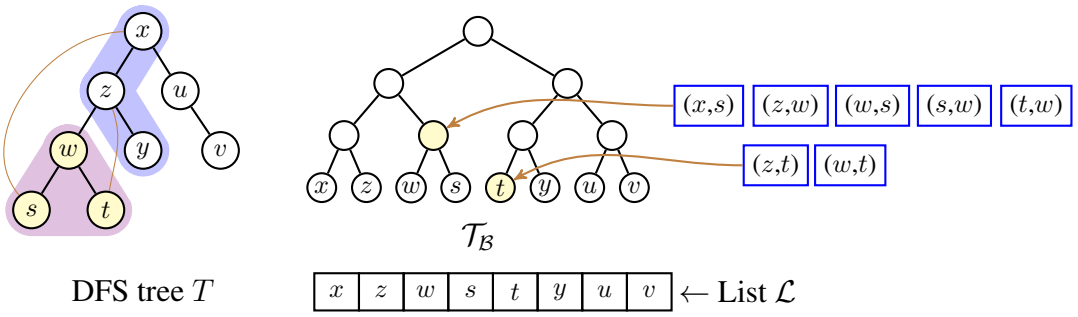

Figure 3.5: (i) The highest edge from subtree $T(w)$ on $path(x, y)$ is edge (x, s) and the lowest edges are edge (z, w) and (z, t). (ii) The vertices of $T(w)$ are represented as union of two subtrees in segment tree $\mathcal{T}_\mathcal{B}$.

To answer $Query(T(w), x, y)$, we thus find the edge closest to x among all the edges reported by the queries $\{Q(z_j, x_i, y_i) | 1 \leq j \leq q \text{ and } 1 \leq i \leq \ell\}$. Thus, $Query(T(w), x, y)$ can be answered in $O(\log^3 n)$ time. Notice that $Query(w, x, y)$ can be considered as a special case where $q = 1$ and $\mathcal{T}_\mathcal{B}(z_1)$ is the leaf node of $\mathcal{T}_\mathcal{B}$ representing w, i.e., $z_1 = w$. The space required by $\mathcal{D}$ is $O(m \log n)$ as each edge is stored at $O(\log n)$ levels in $\mathcal{T}_\mathcal{B}$. Now, the segment tree $\mathcal{T}_\mathcal{B}$ can be built in linear time. Further, for every node $u \in \mathcal{T}_\mathcal{B}$, the sorted list of edges in $\mathcal{E}(u)$ can be computed in linear time by merging the sorted lists of its children. Thus, the binary search tree $\mathcal{E}(u)$ for each node $u \in \mathcal{T}_\mathcal{B}$ can be built in time linear in the number of edges in $\mathcal{E}(u)$. Hence the total time required to build this data structure is $O(m \log n)$. Thus, we have the following theorem.

Theorem 3.3. *The queries $Query(T(w), x, y)$, $Query(w, x, y)$ on T can be answered in $O(\log^3 n)$ worst case time using a data structure $\mathcal{D}$ of size $O(m \log n)$, which can be built in $O(m \log n)$ time.*

Note: Procedure Reroot can also use a simpler version of $\mathcal{D}$ which requires a smaller query time. However, our *generic* algorithm (described in Section 3.7) would require these additional features of $\mathcal{D}$ as follows.

1. For Procedure Reroot, the binary search tree $\mathcal{E}(u)$ stored at each node u of $\mathcal{T}_\mathcal{B}$ can be replaced by an array storing the sorted list of edges, making it simpler to implement. However, our *generic* algorithm also requires deletion of edges from $\mathcal{D}$. An edge can be deleted from $\mathcal{D}$ by deleting the edge from the binary search trees stored at its endpoints and their ancestors in $\mathcal{T}_\mathcal{B}$. Since a deletion in a binary search tree takes $O(\log n)$ time, an edge can be deleted from $\mathcal{D}$ in $O(\log^2 n)$ time.

2. Procedure Reroot only performs the second type of query, i.e., $Query(T(w), x, y)$. Thus, it would essentially be querying only the part of $path(x, y)$ comprising of the ancestors of w in $path(x, y)$. This is thus equivalent to $Query(T(w), LCA(x, w), LCA(y, w))$ (see Figure 3.5), which will answer the required query as the only edges from $T(w)$ in this interval are incident on $path(x, y)$. In such a case, heavy-light decomposition and hence division of $path(x, y)$ to $O(\log n)$ subpaths would not be required. Hence, on each node in $u \in \mathcal{T}_\mathcal{B}$, the query is performed for a single path, requiring total $O(\log^2 n)$ time. However, our *generic* algorithm also uses the first type of query, i.e.,

$Query(w, x, y)$, where w can be an ancestor of x and y. In such a case, we need to perform the query only on contiguous intervals of $\mathcal{L}$ as the interval between x and y in $\mathcal{L}$ would have several other edges from w that are not incident on $path(x, y)$. This necessitates the use of heavy-light decomposition and hence each query requires $O(\log^3 n)$ time.

3.5 Handling multiple updates - Overview

DFS tree can be computed in $\tilde{O}(n)$ time after a single update in the graph, by reducing it to Procedure Reroot. However, the same procedure cannot be directly applied to handle a sequence of updates because of the following reason. The efficiency of Procedure Reroot crucially depends on the data structure $\mathcal{D}$ which is built using the DFS tree T of the original graph. Thus, when the DFS tree is updated, we are required to rebuild $\mathcal{D}$ for the updated tree. Now, rebuilding $\mathcal{D}$ is highly inefficient because it requires $O(m \log n)$ time. Thus, in order to handle a sequence of updates, our aim is to use the same $\mathcal{D}$ for handling multiple updates, without having to rebuild it after every update. We now give an overview of the algorithm that reports the DFS tree after a set U of updates.

In case of a single update, all the edges reported by $\mathcal{D}$ are added to the final DFS tree T^*. However, while handling multiple updates, we use $\mathcal{D}$ to build *reduced adjacency lists* for vertices of the graph, such that the DFS traversal of the graph using these *sparser* lists gives the DFS tree of the updated graph. Now, the data structure $\mathcal{D}$ finds the lowest/highest edge from a subtree of T to an ancestor-descendant path of T. Thus, in order to employ $\mathcal{D}$ to report DFS tree of $G+U$, we need to ensure that the queried subtrees and paths do not contain any failed edges or vertices from U. Hence, for any set U of updates, we compute a partitioning of T into a disjoint collection of ancestor-descendant paths and subtrees such that none of these subtrees and paths contain any failed edge or vertex. An important property of this partitioning is that there are no edges from G lying between any two subtrees in this partitioning. We refer to this partitioning as a *disjoint tree partitioning*. Note that this partitioning depends only upon the vertex and edge failures present in the set U.

Recall that during the DFS traversal we need to find the lowest edge from each component C of the unvisited graph. It turns out that any component C can be represented as a union of subtrees and ancestor-descendant paths of the original DFS tree T. The components property can now be employed to compute the reduced adjacency lists of the vertices of the graph as follows. We just find the lowest edge from each of the subtrees and the ancestor-descendant paths to T^* by querying the data structure $\mathcal{D}$. Let this edge be (x, y) where $x \in T^*$ and $y \in C$. We can just add y to the reduced adjacency list $L(x)$ of x. Since the components property ensures the remaining edges to T^* can be ignored, the DFS traversal would thus consider all possible candidates for the lowest edge from every component C to T^*. Let the initial disjoint tree partitioning consist of a set of ancestor-descendant paths $\mathcal{P}$ and a set of subtrees $\mathcal{T}$. The algorithm for computing a DFS tree of $G + U$ can be summarized as follows:

Perform the static DFS traversal on the graph with the elements of $\mathcal{P} \cup \mathcal{T}$ as the super *vertices. Visiting a super vertex v^* by the algorithm involves extracting an ancestor-descendant path p_0 from v^* and attaching it to the partially grown DFS tree T^*. The remaining part of v^* is added back to $\mathcal{P} \cup \mathcal{T}$ as* new *super vertices. Thereafter, the reduced*

adjacency lists of the vertices on path p_0 are computed using the data structure $\mathcal{D}$. The algorithm then continues to find the next super vertex using the reduced adjacency lists and so on.

3.6 Disjoint Tree Partitioning

We formally define disjoint tree partitioning as follows.

Definition 3.1. *Given a DFS tree T of an undirected graph G and a set U of failed vertices and edges, let A be a vertex set in $G + U$. The disjoint tree partitioning defined by A is a partition of the subgraph of T induced by A into*

1. *A set of paths $\mathcal{P}$ such that (i) each path in $\mathcal{P}$ is an ancestor-descendant path in T and does not contain any deleted edge or vertex, and (ii) $|\mathcal{P}| \leq |U|$.*

2. *A set of trees $\mathcal{T}$ such that each tree $\tau \in \mathcal{T}$ is a subtree of T which does not contain any deleted edge or vertex.*

Note that for any $\tau_1, \tau_2 \in \mathcal{T}$, there is no edge between τ_1 and τ_2 because T is a DFS tree.

The disjoint tree partitioning for set $A = V \setminus \{r\}$ can be computed as follows. Let V_f and E_f respectively denote the set of failed vertices and edges associated with the updates U. We initialize $\mathcal{P} = \emptyset$ and $\mathcal{T} = \{T(w) \mid w \text{ is a child of } r\}$. We refine the partitioning by processing each vertex $v \in V_f$ as follows (see Figure 3.6 (i)).

- If v is present in some $T' \in \mathcal{T}$, we add the path from $par(v)$ to the root of T' to $\mathcal{P}$. We remove T' from $\mathcal{T}$ and add all the subtrees hanging from this path to $\mathcal{T}$.

- If v is present in some path $p \in \mathcal{P}$, we split p at v into two paths. We remove p from $\mathcal{P}$ and add these two paths to $\mathcal{P}$.

Edge deletions are handled as follows. We first remove edges from E_f that don't appear in T. Processing of the remaining edges from E_f is quite similar to the processing of V_f as described above. For each edge $e \in E_f$, just visualize deleting an imaginary vertex lying at mid-point of the edge e (see Figure 3.6 (ii)). It takes $O(n)$ time to process any $v \in V_f$ and any $e \in E_f$.

Note that each update can add at most one path to $\mathcal{P}$. So the size of $\mathcal{P}$ is bounded by $|U|$. The fact that T is a DFS tree of G ensures that no two subtrees in $\mathcal{T}$ will have an edge between them. So $\mathcal{P} \cup \mathcal{T}$ satisfies all the conditions stated in Definition 3.1.

Lemma 3.6.1. *Given an undirected graph G with a DFS tree T and a set U of failing vertices and edges, we can find a disjoint tree partition of set $V \setminus \{r\}$ in $O(n|U|)$ time.*

3.7 Fault tolerant DFS Tree

We first present a fault tolerant algorithm for a DFS tree. Let U be any given set of failed vertices or edges in G. In order to compute the DFS tree T^* for $G + U$, our algorithm first constructs a disjoint tree partition $(\mathcal{T}, \mathcal{P})$ for $V \setminus \{r\}$ defined by the updates U (see Lemma 3.6.1). Thereafter, it can be visualized as the static DFS traversal on the graph

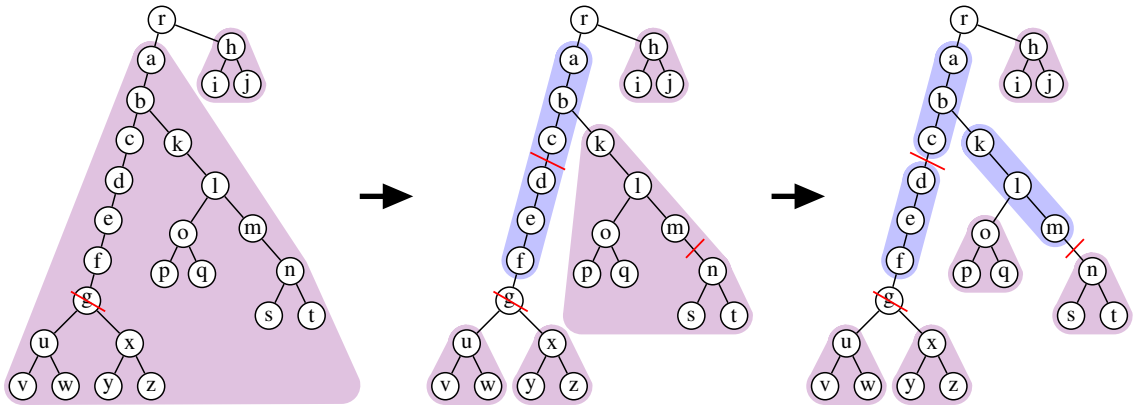

Figure 3.6: Disjoint tree partitioning for $V \setminus \{r\}$: (i) Initializing $\mathcal{T} = \{T(a), T(h)\}$ and $\mathcal{P} = \emptyset$, (ii) Disjoint tree partition obtained after deleting the vertex g. (iii) Final disjoint tree partition obtained after deleting the edges (c, d) and (m, n).

whose (*super*) vertices are the elements of $\mathcal{P} \cup \mathcal{T}$. Note that our notion of super vertices is for the sake of understanding only.

Consider the stack-based implementation of the static algorithm for computing a DFS tree rooted at a vertex r in graph G (refer to Figure 3.7(i)). Our algorithm for computing DFS tree for $G + U$ (refer to Figure 3.7(ii)) is quite similar to the static algorithm. The only points of difference are the following.

- In the static DFS algorithm whenever a vertex is visited, it is attached to the DFS tree and pushed into the stack S. In our algorithm when a vertex u in some super vertex $v_s \in \mathcal{P} \cup \mathcal{T}$ is visited, a path starting from u is extracted from v_s and attached to the DFS tree, and this entire path is pushed into the stack S.

- Instead of scanning the entire adjacency list $N(w)$ of a vertex w, the reduced adjacency list $L(w)$ is scanned.

When a path is extracted from a super vertex v_s, the remaining unvisited part of v_s is added back to $\mathcal{T} \cup \mathcal{P}$. However, we need to ensure that the properties of disjoint tree partitioning are satisfied in the updated $\mathcal{T} \cup \mathcal{P}$. This is achieved using Procedure DFS-in-Path and Procedure DFS-in-Tree, which also build the reduced adjacency list for the vertices on the path. The construction of a sparse reduced adjacency list is inspired by the components property which can be *adapted* in the context of our algorithm as follows.

Lemma 3.7.1 (Adapted components property)**.** *When a path p is attached to the partially constructed DFS tree T^* during the algorithm, for every edge (x, y), where $x \in p$ and y belongs to the unvisited graph the following condition holds. Either y is added to $L(x)$ or y' is added to $L(x')$ for some edge (x', y') where x' is a descendant (not necessarily proper) of x in p and y' is connected to y in the unvisited graph.*

We now describe how the properties of disjoint tree partitioning and hence the adapted components property are maintained by our algorithm when a vertex $v \in v_s$ is visited by the traversal.

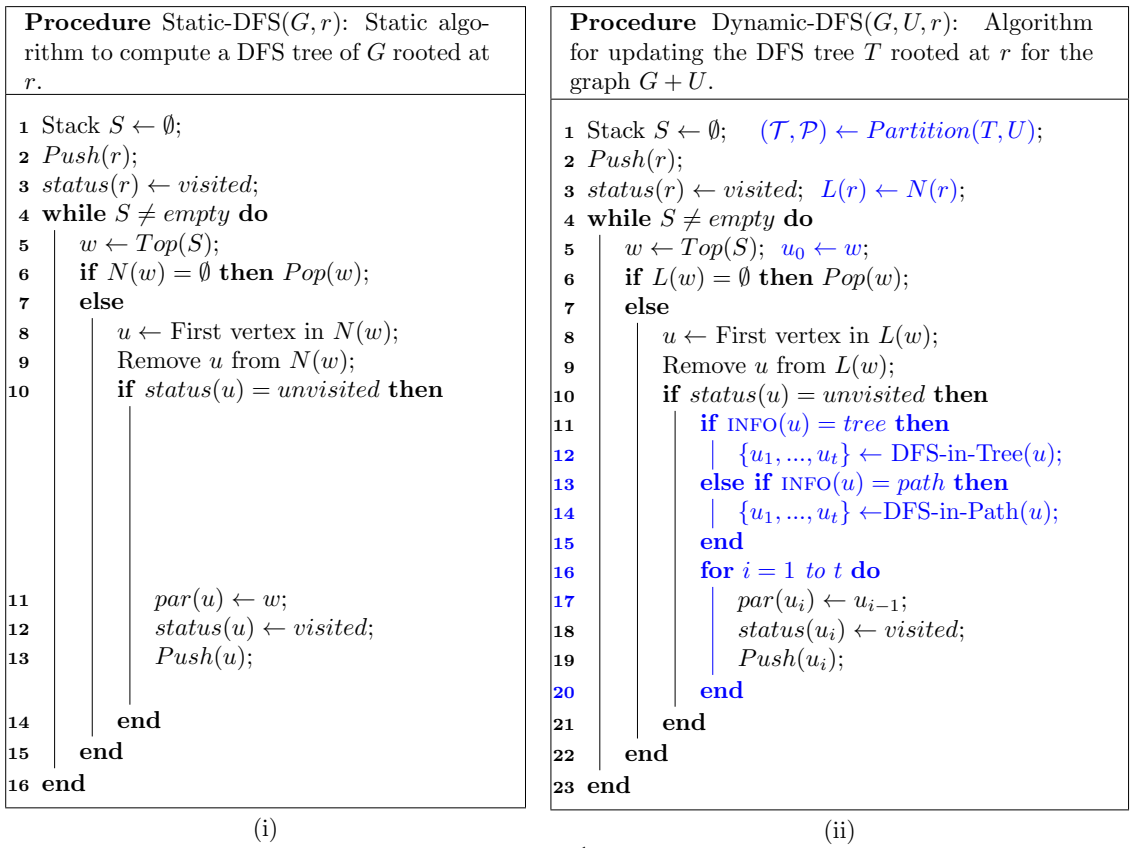

Figure 3.7: The static (and dynamic) algorithm for computing (updating) a DFS tree. The key differences are shown in blue.

1. Let $v_s = path(x, y) \in \mathcal{P}$. Exploiting the flexibility of DFS, we traverse from v to the farther end of $path(x, y)$. Now, $path(x, y)$ is removed from $\mathcal{P}$ and the untraversed part of $path(x, y)$ (with length at most half of $|path(x, y)|$) is added back to $\mathcal{P}$. We refer to this as *path halving*. This technique was also used by Aggarwal and Anderson [AA88] in their parallel algorithm for computing DFS tree in undirected graphs. Notice that $|\mathcal{P}|$ remains unchanged or decreases by 1 after this step.

2. Let $v_s = \tau \in \mathcal{T}$. Exploiting the flexibility of a DFS traversal, we traverse the path from v to the root of τ, say x, and add it to T^*. Thereafter, τ is removed from $\mathcal{T}$ and all the subtrees hanging from this path are added to $\mathcal{T}$. Observe that every newly added subtree is also a subtree of the original DFS tree T. So the properties of disjoint tree partitioning are satisfied after this step as well.

Let $path(v, x)$ be the path extracted from v_s. For each vertex w in this newly added path, we compute $L(w)$ ensuring the adapted components property as follows.

(i) For each path $p \in \mathcal{P}$, among potentially many edges incident on w from p, we just add any one edge.

(ii) For each tree $\tau' \in \mathcal{T}$, we add at most one edge to L as follows. Among all edges incident on τ' from $path(v, x)$, if (w, z) is the edge such that w is nearest to x on

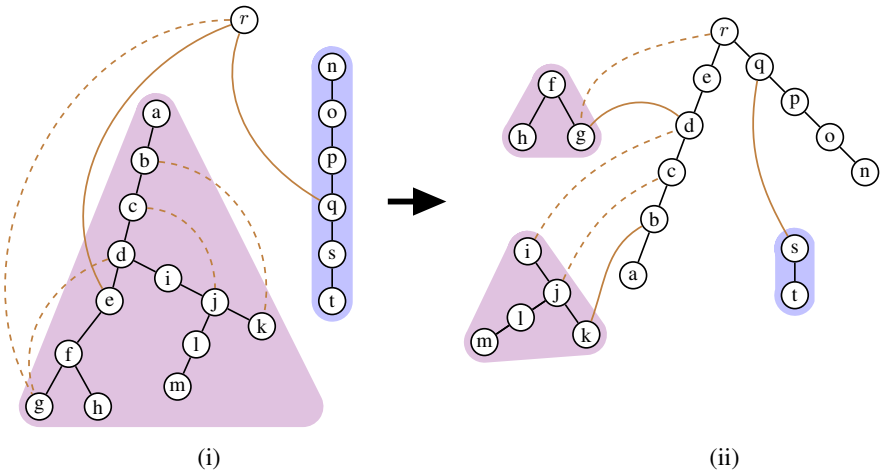

Figure 3.8: Visiting a super vertex from $\mathcal{T} \cup \mathcal{P}$. (i) The algorithm visits $T(a) \in \mathcal{T}$ using the edge (r, e) and the $path(n, t) \in \mathcal{P}$ using the edge (r, q). (ii) Traversal extracts $path(e, a)$ and $path(q, n)$ and augment it to T^*. The unvisited segments are added back to $\mathcal{T}$ and $\mathcal{P}$.

$path(v, x)$, then we add z to $L(w)$. However, for the case $v_s \in \mathcal{T}$, we have to consider only the newly added subtrees in $\mathcal{T}$ for this step. This is because the disjoint tree partitioning ensures the absence of edges between v_s and any other tree in $\mathcal{T}$.

Figure 3.8 provides an illustration of how $\mathcal{T} \cup \mathcal{P}$ is updated when a super vertex in $\mathcal{T} \cup \mathcal{P}$ is visited.

3.7.1 Implementation of our Algorithm

We now describe our algorithm in full detail. Firstly we delete all the failed edges in U from the data structure $\mathcal{D}$. Now, the algorithm begins with a disjoint tree partition $(\mathcal{T}, \mathcal{P})$ which evolves as the algorithm proceeds. The state of any unvisited vertex in this partition is captured by the following three variables.
-INFO(u): this variable is set to *tree* if u belongs to a tree in $\mathcal{T}$, and set to *path* otherwise
-ISROOT(v): this variable is set to $True$ if v is the root of a tree in $\mathcal{T}$, and $False$ otherwise.
-PATHPARAM(v): if v belongs to some path, say $path(x, y)$, in $\mathcal{P}$, then this variable stores the pair (x, y), and *null* otherwise.

Procedure Dynamic-DFS : For each vertex v, $status(v)$ is initially set as *unvisited*, and $L(v)$ is initialized to $\emptyset$. First a disjoint tree partition is computed for the DFS tree T based on the updates U. The procedure Dynamic-DFS then inserts the root vertex r into the stack S. While the stack is non-empty, the procedure repeats the following steps. It reads the top vertex from the stack. Let this vertex be w. If $L(w)$ is empty then w is popped out from the stack, else let u be the first vertex in $L(w)$. If vertex u is unvisited till now, then depending upon whether u belongs to some tree in $\mathcal{T}$ or some path in $\mathcal{P}$, Procedure DFS-in-Tree or DFS-in-Path is executed. A path p_0 is then returned to Procedure Dynamic-DFS where for each vertex of p_0 parent is assigned and status is marked visited. The whole of this path is then pushed into stack. The procedure proceeds to the

Procedure DFS-in-Tree(u): DFS traversal enters from node u and exits from v, the root of the tree containing node u in set $\mathcal{T}$.

```
1  v ← u;
2  while IsRoot(v) ≠ True do
3  |  v ← par(v)
4  end
5  IsRoot(v) ← False;
6  T ← T\T(v);
7  (w_1, ..., w_t) ← path(u, v);
8  for i = 1 to t do
9  |  foreach path(x, y) ∈ P do
10 |  |  if Query(w_i, x, y) ≠ ∅ then
11 |  |  |  (w_i, z) ← Query(w_i, x, y);
12 |  |  |  L(w_i) ← L(w_i) ∪ {z};
13 |  |  end
14 |  end
15 |  foreach child w of w_i except w_{i-1} do
16 |  |  (y, z) ← Query(T(w), v, u);
   |  |  /* where y ∈ path(u, v)          */
17 |  |  L(y) ← L(y) ∪ {z};
18 |  |  T ← T ∪ T(w);
19 |  |  IsRoot(w) ← True;
20 |  end
21 end
22 Return path(u, v);
```

Procedure DFS-in-Path(u): DFS traversal enters from node u and exits from v, the farther end of path containing node u in set $\mathcal{P}$.

```
1  (v, d) ← PathParam(u);
2  if dist_T(u, d) > dist_T(u, v) then Swap(v, d);
3  c ← Neighbor of u on path(v, d) nearer to d;
4  P ← (P\path(v, d)) ∪ path(c, d);
5  for c' ∈ path(c, d) do
6  |  PathParam(c') ← (c, d);
7  end
8  (w_1, ..., w_t) ← path(u, v);
9  for i = 1 to t do
10 |  foreach path(x, y) ∈ P do
11 |  |  if Query(w_i, x, y) ≠ ∅ then
12 |  |  |  (w_i, z) ← Query(w_i, x, y);
13 |  |  |  L(w_i) ← L(w_i) ∪ {z};
14 |  |  end
15 |  end
16 end
17 foreach T(w) ∈ T do
18 |  if Query(T(w), v, u) ≠ ∅ then
19 |  |  (y, z) ← Query(T(w), v, u);
   |  |  /* where y ∈ path(u, v)          */
20 |  |  L(y) ← L(y) ∪ {z};
21 |  end
22 end
23 Return path(u, v);
```

Figure 3.9: The pseudocode of Procedures DFS-in-Tree and Procedures DFS-in-Path.

next iteration of While loop with the updated stack.

Procedure DFS-in-Tree : Let vertex u be present in tree, say $T(v)$, in $\mathcal{T}$ (the vertex v can be found easily by scanning the ancestors of u and checking their value of IsRoot). The DFS traversal enters the tree from u and leaves from the vertex v. Let $path(u, v) = \langle w_1 = u, w_2 \ldots, w_t = v \rangle$. The $path(u, v)$ is pushed into stack and attached to the partially constructed DFS tree T^*. We now update the partition $(\mathcal{P}, \mathcal{T})$ and also update the reduced adjacency list for each w_i present on $path(u, v)$ as follows.

1. For each vertex w_i and every path $path(x, y) \in \mathcal{P}$, we perform $Query(w_i, x, y)$ on the data structure $\mathcal{D}$ that returns an edge (w_i, z) such that $z \in path(x, y)$. We add z to $L(w_i)$.

2. Recall that since subtrees in T do not have any cross edge between them, therefore, there cannot be any edge incident on $path(u, v)$ from trees which are already present in $\mathcal{T}$. An edge can be incident only from the subtrees which were hanging from $path(u, v)$. $T(v)$ is removed from $\mathcal{T}$ and all the subtrees of $T(v)$ hanging from $path(u, v)$ are inserted into $\mathcal{T}$. For each such subtree, say τ, inserted into $\mathcal{T}$, we perform $Query(\tau, v, u)$ on the data structure $\mathcal{D}$ that returns an edge, say (y, z), such that $z \in \tau$ and y is nearest to v on $path(u, v)$. We insert z into $L(y)$.

Procedure DFS-in-Path : Let vertex u visited by the DFS traversal lies on a $path(v, y) \in$

$\mathcal{P}$. Assume $dist_T(u, v) > dist_T(u, y)$. The DFS traversal travels from u to v (the farther end of the path). The path $path(v, y)$ in set $\mathcal{P}$ is replaced by its subpath that remains unvisited. The reduced adjacency list of each $w \in path(u, v)$ is updated in a similar way as in the procedure DFS-in-Tree except that in step 2, we perform $Query(\tau, u, v)$ for each $\tau \in \mathcal{T}$. Note that while performing step 1, the vertex w_i can be an ancestor of the vertices of $path(x, y)$. This is because the vertices of a path in $\mathcal{P}$ can be ancestors of the vertices of another path in $\mathcal{P}$. This was not true for Procedure DFS-in-Trees because vertices of a subtree in $\mathcal{T}$ cannot be ancestors of vertices of any path in $\mathcal{P}$. Thus, our data structure $\mathcal{D}$ needs to support queries where w_i is an ancestor of the queried path (refer to the note at the end of Section 3.4).

The reader may refer to Figure 3.9 for pseudocode of Procedures DFS-in-Tree and DFS-in-Path. This completes the description of the fault tolerant algorithm for computing a DFS tree. This algorithm maintains the adapted components property at each stage by construction given that the properties of disjoint tree partitioning are satisfied.

3.7.2 Correctness

It can be seen that the following two invariants hold for the while loop in the Procedure Static-DFS described in Figure 3.7 (i). It is easy to see that these invariants imply the correctness of the algorithm, i.e., the generated tree is a rooted spanning tree where every non-tree edge is a back edge.

I_1: The sequence of vertices in the stack from bottom to top constitutes an ancestor-descendant path from r in the DFS tree computed.

I_2: For each vertex v that is popped out, all vertices in the set $N(v)$ have already been visited.

These two invariants I_1 and I_2 also hold for Procedure Dynamic-DFS described in Figure 3.7 (ii) as follows. Invariant I_1 holds by construction as described in our algorithm. Following lemma proves that invariant I_2 is maintained by our algorithm since it follows the adapted components property by construction.

Lemma 3.7.2. *If the adapted components property is maintained by the Procedure Dynamic-DFS, then invariant I_2 will hold true at each stage of the algorithm.*

Proof. We give a proof by contradiction as follows. Assume that x is the first vertex that is popped out of the stack before some vertex $y \in N(x)$ is visited. Consider the time when a path p containing x was pushed in the stack. Clearly $y \notin L(x)$, hence using the adapted components property we know that some $y' \in L(x')$ is connected to y in the unvisited graph where x' is a descendant (not necessarily proper) of x in p. Let p^* be a path between y' and y in the unvisited graph.

Now, consider the time when x is popped out of the stack. Clearly all its descendants including x' have been popped out, so using invariant I_2 for x', y' has been visited by the traversal. Thus, p^* can be divided into two non-empty sets A and B denoting visited and unvisited vertices of p^* respectively. Here $y' \in A$ and $y \in B$, thus clearly for the last vertex of p^* that is present in A, the invariant I_2 is not satisfied. This contradicts our assumption that x is the first vertex that is popped out of the stack for which I_2 is not satisfied. Thus, maintenance of the adapted components property ensures the invariant I_2 in our algorithm. □

Hence, our algorithm indeed computes a valid DFS tree for $G + U$.

3.7.3 Time complexity analysis

As described earlier the disjoint tree partitioning and the components property play a key role in the efficiency of our algorithm. They allow us to limit the size of the reduced adjacency lists L, that are built during the algorithm. Our algorithm computes T^* by performing a DFS traversal on the reduced adjacency list L. Thus, the time complexity of our algorithm is $O(n + |L|)$ excluding the time required to compute L.

We first establish a bound on the size of L. In each step our algorithm extracts a path from $v_s \in \mathcal{P} \cup \mathcal{T}$ and attaches it to T^*. Let P_t and P_p denote the set of such paths that originally belonged to some tree in $\mathcal{T}$ and some path in $\mathcal{P}$, respectively. For every path $p_0 \in P_t \cup P_p$ our algorithm performs the following queries on $\mathcal{D}$.

(i) For each vertex w in p_0, we query each path in $\mathcal{P}$ for an edge incident on the vertex w. Thus, the total number of edges added to L by these queries is $O(n|\mathcal{P}|)$.

(ii) If p_0 belongs to P_p, then we query for an edge from each $\tau \in \mathcal{T}$ to p_0. It follows from the path halving technique that each path in $\mathcal{P}$ reduces to at most half of its length whenever some path is extracted from it and attached to T^*. Hence, the size of P_p is bounded by $|\mathcal{P}| \log n$.

(iii) If p_0 belongs to P_t, then we query for an edge from only those subtrees which were hanging from p_0. Note that these subtrees will now be added to set $\mathcal{T}$. Hence, the total number of trees queried for this case will be bounded by number of trees inserted to $\mathcal{T}$. Since each subtree can be added to $\mathcal{T}$ only once, these edges are bounded by $O(n)$ throughout the algorithm.

Thus, the size of L is bounded by $O\big(n(1 + |\mathcal{P}|) \log n\big)$. Since each edge added to L requires querying the data structure $\mathcal{D}$ which takes $O(\log^3 n)$ time, the total time taken to compute L is $O\big(n(1 + |\mathcal{P}| \log n) \log^3 n\big)$. Thus, we have the following lemma.

Lemma 3.7.3. *An undirected graph can be preprocessed to build a data structure of $O(m \log n)$ size such that for any set U of k failed vertices or edges (where $k \leq n$), the DFS tree of $G + U$ can be reported in $O(n(1 + |\mathcal{P}| \log n) \log^3 n)$ time.*

From Definition 3.1 we have that $|\mathcal{P}|$ is bounded by $|U|$. Thus, we have the following theorem.

Theorem 3.4. *An undirected graph can be preprocessed to build a data structure of $O(m \log n)$ size such that for any set U of k failed vertices or edges (where $k \leq n$), the DFS tree of $G + U$ can be reported in $O(nk \log^4 n)$ time.*

It can be observed that Theorem 3.4 directly implies a data structure for fault tolerant DFS tree.

3.7.4 Extending the algorithm to handle insertions

In order to update the DFS tree, our focus has been to restrict the number of edges that are processed. For the case when the updates are deletions only, we have been able to

restrict this number to $O(nk \log n)$, for a given set of k updates (failure of vertices or edges). We now describe the procedure to handle vertex and edge insertions. Let V_I be the set of vertices inserted, and E_I be the set of edges inserted. (including the edges incident to the vertices in V_I). If there are k vertex insertions, the size of E_I is bounded by nk. So even if we add all the edges in E_I to the reduced adjacency lists, the size of L would still be bounded by $O(nk \log n)$. Hence, we perform the following two additional steps before starting the DFS traversal.

- Initialize $L(v)$ to store the edges in E_I instead of $\emptyset$. That is, $L(v) \leftarrow \{y \mid (y, v) \in E_I\}$
- Each newly inserted vertex is treated as a singleton tree and added to $\mathcal{T}$. That is, $\mathcal{T} \leftarrow \mathcal{T} \cup \{x | x \in V_I\}$.

In order to establish that our algorithm, after incorporating the insertions, correctly computes a DFS tree of $G + U$, we need to ensure that all the edges *essential* for DFS traversal as described in the adapted components property are added to L. All the essential edges from G are added to L during the algorithm itself. In case an essential edge belongs to E_I, the edge has already been added to L during its initialization. Note that the time taken by our algorithm remains unchanged since the size of L remains bounded by $O(nk \log n)$. This completes the proof of our main result stated as follows.

Theorem 3.5. *An undirected graph can be preprocessed to build a data structure of $O(m \log n)$ size such that for any set U of $k \leq n$ updates, a DFS tree of $G + U$ can be reported in $O(nk \log^4 n)$ time.*

Let us consider the case when U consists of insertions only. In this case $\mathcal{P}$ will be an empty set. As discussed above, we initialize the reduced adjacency lists using E_I whose size is equal to $|U|$. Additionally, since the vertices in V_I would be added to the set of trees, $|V_I|$ would be added to n. Hence, Lemma 3.7.3 implies the following theorem.

Theorem 3.6. *An undirected graph can be preprocessed to build a data structure of $O(m \log n)$ size such that for any set U of k vertex insertions and m' edge insertions, a DFS tree of $G + U$ can be reported in $O(m' + (n + k) \log^3 n)$ time.*

Note: In Theorem 3.6, the size of input is $k + m'$. Also, even a single insertion may change $\Omega(n)$ edges of the DFS tree. Hence our algorithm is optimal up to $\tilde{O}(1)$ factors for processing edge or vertex insertions if the DFS tree has to be maintained explicitly.

3.8 Fully dynamic DFS

We now describe the overlapped periodic rebuilding technique to convert our algorithm for computing a DFS tree after k updates to fully dynamic and incremental algorithms for maintaining a DFS tree. Similar technique was used by Thorup [Tho05] for maintaining fully dynamic all pairs shortest paths.

In the fully dynamic model, we need to report the DFS tree after every update in the graph. Given the data structure $\mathcal{D}$ built using the DFS tree of the graph G, we are able to report the DFS tree of $G + U$ after $|U| = k$ updates in $\tilde{O}(nk)$ time. This becomes inefficient if k becomes large. Rebuilding $\mathcal{D}$ after every update is also inefficient as it takes $\tilde{O}(m)$ time to build $\mathcal{D}$. Thus, it is better to rebuild $\mathcal{D}$ after every $|U'| = c$ updates for a

carefully chosen c. Let $\mathcal{D}'$ be the data structure built using the DFS tree of the updated graph $G + U'$ with $|U'| = c$. $\mathcal{D}'$ can thus be used to process the next c updates efficiently (see Figure 3.10 (a)). The cost of building $\mathcal{D}'$ can thus be amortized over these c updates.

To achieve an efficient worst case update time, we divide the building of $\mathcal{D}'$ over the first c updates. This $\mathcal{D}'$ is then used by our algorithm in the next c updates, during which a new $\mathcal{D}''$ is built in a similar manner and so on (see Figure 3.10 (b)). The following lemma describes how this technique can be used in general for any dynamic graph problem. For notational convenience we denote any function $f(m, n)$ as f.

Lemma 3.8.1. *Let D be a data structure that can be used to report the solution of a graph problem after a set of U updates on an input graph G. If D can be build in $O(f)$ time and the solution for graph $G + U$ can be reported in $O(h + |U| \times g)$ time, then D can be used to report the solution after every update in worst case $O(\sqrt{fg} + h)$ update time, given that $\sqrt{f/g} \leq n$.*

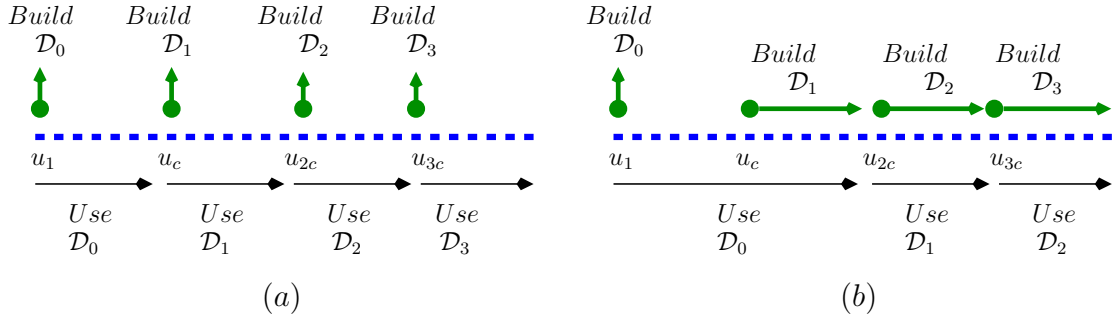

Figure 3.10: (a) Fully dynamic algorithm with amortized update time. (b) De-amortization of the algorithm.

Proof. We first present an algorithm that achieves amortized $O(\sqrt{fg} + h)$ update time. It is based on the simple idea of periodic rebuilding. Given the input graph G_0 we preprocess it to compute the data structure D_0 over it. Now, let $u_1, ..., u_c$ $(c \leq n)$ be the sequence of first c updates on G_0. To report the solution after i^{th} update we use D_0 to compute the solution for $G_0 + \{u_1, ..., u_i\}$. This takes $O(h + (i \times g))$ time. So the total time for preprocessing and handling the first c updates is $O(f + \sum_{i=1}^{c} h + (i \times g))$. Therefore, the average time for the first c updates is $O(f/c + c \times g + h)$. Minimizing this quantity over c gives the optimal value $c_0 = \sqrt{f/g}$ which is bounded by n. So, after every c_0 updates we rebuild our data structure and use it for the next c_0 updates (see Figure 3.10(a)). Substituting the value of c_0 gives the amortized time complexity as $O(\sqrt{fg} + h\ )$.

The above algorithm can be de-amortized as follows. Let $G_1, G_2, G_3, \ldots$ be the sequence of graphs obtained after $c_0, 2c_0, 3c_0, ..$ updates. We use the data structure D_0 built during preprocessing to handle the first $2c_0$ updates. Also, after the first c_0 updates we start building the data structure D_1 over G_1. This D_1 is built in c_0 steps, thus the extra time spent per update is $f/c_0 = O(\sqrt{fg})$ only. We use D_1 to handle the next c_0 updates on graph G_2, and also in parallel compute the data structure D_2 over the graph G_2. (See Figure 3.10(b)). Since the time for building each data structure is now divided in c_0 steps, we have that the worst case update time as $O(\sqrt{fg} + h\ )$. □

The above lemma combined with Theorems 3.5 and 3.6 directly implies the following results for the fully dynamic DFS tree problem and the incremental DFS tree problem, respectively.

(For the following theorem we use Theorem 3.5, implying $f = m \log n$, $g = n \log^4 n$ and $h = 0$.)

Theorem 3.7. *There exists a fully dynamic algorithm for maintaining a DFS tree in an undirected graph that uses $O(m \log n)$ preprocessing time and can report a DFS tree after each update in the worst case $O(\sqrt{mn} \log^{2.5} n)$ time. An update in the graph can be insertion / deletion of an edge as well as a vertex.*

(For the following theorem we use Theorem 3.6, implying $f = m \log n$, $g = \log^3 n$ and $h = n \log^3 n$.)

Theorem 3.8. *There exists an incremental algorithm for maintaining a DFS tree in an undirected graph that uses $O(m \log n)$ preprocessing time and can report a DFS tree after each edge insertion in the worst case $O(n \log^3 n)$ time.*

3.9 Applications

Our fully dynamic algorithm for maintaining a DFS tree can be used to solve various dynamic graph problems such as dynamic subgraph connectivity, biconnectivity and 2-edge connectivity . Note that these problems are solved trivially using a DFS tree in the static setting. Let us now describe the importance of our result in the light of the existing results for these problems.

Existing Results

The dynamic subgraph connectivity problem is defined as follows. Given an undirected graph, the status of any vertex can be switched between *active* and *inactive* in an update. For any online sequence of updates interspersed with queries, the goal is to efficiently answer each connectivity queries on the subgraph induced by the active vertices. This problem can be solved by using dynamic connectivity data structures [EGIN97, Fre85, HdLT01, KKM13] that answer connectivity queries under an online sequence of edge updates. This is because switching the state of a vertex is equivalent to $O(n)$ edge updates. Chan [Cha06] introduced this problem and showed that it can be solved more efficiently. He gave an algorithm using FMM (fast matrix multiplication) that achieves $O(m^{0.94})$ amortized update time and $\tilde{O}(m^{1/3})$ query time. Later Chan et al. [CPR08] presented a new algorithm that improves the amortized update time to $\tilde{O}(m^{2/3})$. They also mentioned the following among the open problems.

1. Is it possible to achieve constant query time with worst case sublinear ($o(m)$) update time ?

2. Can non trivial updates be obtained for richer queries such as counting the number of connected components ?

Duan [Dua10] partially answered the first question affirmatively but at the expense of a much higher update time and non-constant query time. He presented an algorithm

References	**Update Time**	**Query Time**
Frederickson [Fre85] (1985), Eppstein et al. [EGIN97] (1997)	$O(n\sqrt{n})$	$O(1)$
Holm et al. [HdLT01] (2001)	$\tilde{O}(n)$ amortized	$\tilde{O}(1)$
Chan [Cha06] (2006)	$\tilde{O}(m^{0.94})$ amortized	$\tilde{O}(m^{1/3})$
Chan et al. [CPR08] (2008)	$\tilde{O}(m^{2/3})$ amortized	$\tilde{O}(m^{1/3})$
Duan [Dua10] (2010)	$\tilde{O}(m^{4/5})$	$\tilde{O}(m^{1/5})$
Kapron et al. [KKM13] (2013)	$\tilde{O}(n)$	$\tilde{O}(1)$ (Monte Carlo)
New	$\tilde{O}(\sqrt{mn})$	$O(1)$

Table 3.1: Current-state-of-the-art of the algorithms for the dynamic subgraph connectivity.

with $O(m^{4/5})$ worst case update time and $O(m^{1/5})$ query time, improving the worst case bounds for the problem. Kapron et al. [KKM13] presented a randomized algorithm for fully dynamic connectivity which takes $\tilde{O}(1)$ time per update and answers the query correctly with high probability in $\tilde{O}(1)$ time, giving a Monte Carlo algorithm for subgraph connectivity with worst case $\tilde{O}(n)$ update time. Thus, their result answered the first question in a randomized setting. However, in the deterministic setting both these questions were still open. Our result answers both these questions affirmatively for the deterministic setting as well. Our fully dynamic DFS algorithm directly provides an $\tilde{O}(\sqrt{mn})$ update time and $O(1)$ query time algorithm for the dynamic subgraph connectivity problem. Our algorithm maintains the number of connected components simply as a byproduct. In fact, our fully dynamic DFS algorithm solves a generalization of dynamic subgraph connectivity - in addition to just switching the status of vertices, it allows insertion of new vertices as well. Hence the existing results offer different trade-offs between the update time and the query time, and differ on the types (amortized or worst case) of update time and the types (deterministic or randomized) of query time. Our algorithm, in particular, improves the deterministic worst case bounds for the problem (see Figure 3.1). Further, unlike all the previous algorithms for dynamic subgraph connectivity, which use heavy machinery of existing dynamic algorithms, our algorithm is arguably much simpler and self contained.

Exploiting the rich structure of DFS trees, we also obtain $\tilde{O}(\sqrt{mn})$ update time algorithms for dynamic biconnectivity and dynamic 2-edge connectivity under vertex updates in a seamless manner. These problems have mainly been studied in the dynamic setting under edges updates. Some of these results also allow insertion and deletion of isolated vertices. Our result, on the other hand does not impose any such restriction on insertion or deletion of vertices. Figure 3.2 illustrates our results and the existing results in the right perspective. We now describe how our algorithm can be used to solve these problems.

3.9.1 Algorithm

The solution of dynamic subgraph connectivity follows seamlessly from our fully dynamic algorithm as follows. As mentioned in Appendix A, we maintain a DFS tree rooted at a dummy vertex r, such that the subtrees hanging from its children corresponds to the

References	**Update Time**	**Query Time**
Frederickson [Fre85] (1985), Eppstein et al. [EGIN97] (1997)†	$O(n\sqrt{n})$	$O(1)$
Henzinger [Hen00] (2000) $*$	$\tilde{O}(n\sqrt{n})$	$O(1)$
Holm et al. [HdLT01] (2001) $*\dagger$	$\tilde{O}(n)$ amortized	$\tilde{O}(1)$
New $*\dagger$	$\tilde{O}(\sqrt{mn})$	$O(1)$

Table 3.2: Current-state-of-the-art of the algorithms for the dynamic biconnectivity ($*$) and dynamic 2-edge connectivity (†) under vertex updates.

connected components of the graph. Hence, the connectivity query for any two vertices can be answered by comparing their ancestors at depth two (i.e. children of r). This information can be stored for each vertex and updated whenever the DFS tree is updated. Thus, we have a data structure for subgraph connectivity with worst case $\tilde{O}(\sqrt{mn})$ update time and $O(1)$ query time. Our fully dynamic DFS algorithm can be extended to solve fully dynamic biconnectivity and 2-edge connectivity under vertex updates as follows.

A set S of vertices in a graph is called a *biconnected component* if it is a maximal set of vertices such that on failure of any vertex w in S, the vertices of $S\setminus\{w\}$ remains connected. Similarly, a set S is said to be *2-edge connected component* if it is a maximal set of vertices such that the failure of any edge with both endpoints in S does not disconnect any two vertices in S. The biconnectivity and 2-edge connectivity queries can be answered easily by finding *articulation points* and *bridges* of the graph. It can be shown [CLRS09] that two vertices belong to same biconnected component if and only if the path connecting them in a DFS tree of the graph does not pass through any *articulation point*. Similarly, two vertices belong to same 2-edge connected component if and only if the path connecting them in a DFS tree of the graph does not have a *bridge*. An articulation point and a bridge of a graph can be defined as follows:

Definition 3.2. *Given a graph $G = (V, E)$, a vertex $v \in V$ is called an articulation point of G if there exist a pair of vertices $x, y \in V$ such that every path between x and y in G passes through v.*

Definition 3.3. *Given a graph $G = (V, E)$, an edge $e \in E$ is called a bridge of G if there exist a pair of vertices $x, y \in V$ such that every path between x and y in G passes through e.*

The articulation points and bridges of a graph can be easily computed by using DFS traversal of the graph. Given a DFS tree T of an undirected graph G, we can index the vertices in the order they are visited by the DFS traversal. This index is called the *DFN number* of the vertex. The *high number* of a vertex v is defined as the lowest DFN number vertex from which there is an edge incident to $T(v)$. Now, any non-root vertex v will be an articulation point of the graph if high number of at least one of its children is equal to $DFN(v)$. The root r of the DFS tree T will be an articulation point if it has more than one child. An edge (x, y) of the DFS tree, where $x = par(y)$, will be a bridge if the high number of y is $DFN(x)$ and the high number of each child of y (if any) is equal to

$DFN(y)$. Thus, given the high number of each vertex in the DFS tree, the articulation points and bridges can be determined in $O(n)$ time.

We can augment our fully dynamic DFS algorithm with an additional procedure to compute high number of each vertex using the same time bounds. For this we show that given any set of k updates to graph G, while computing the new tree T^* we also compute the high number of each vertex in $O(nk \log^4 n)$ time. For each vertex x, let $a(x)$ denote the highest ancestor of x in T^* such that $(x, a(x))$ is an edge in $G + U$. Note that if $(x, a(x))$ is a newly added edge, then it can be easily computed by scanning all the new edges added to the graph. This is due to fact that the total number of new edges added to G is bounded by nk. So we restrict ourselves to the case when $(x, a(x))$ was originally present in the graph G. Recall that our algorithm computes T^* by attaching paths to the partially grown tree. Let P_t and P_p be the set of paths attached to T^* (during its construction) that originally belonged to $\mathcal{T}$ and $\mathcal{P}$ respectively. Further, path halving ensures that the size of P_p is bounded by $k \log n$. For each path $p_0 \in P_t \cup P_p$, let $H(p_0)$ denote the vertex in p_0 that is closest to r in T^*.

We now present the procedure for constructing a subset $A(x)$ of neighbors of x while computing T^* in $O(nk \log^4 n)$ time, such that the following condition holds.

- For a vertex x, if $a(x) \notin A(x)$, then there is some descendant y of x in T^* such that $a(x) \in A(y)$.

It is easy to see that if we get such an $A(x)$ for each x, then high number of each vertex can be computed easily by processing the vertices of T^* in bottom-up manner. Now, depending upon whether paths containing x and $a(x)$ belong to set P_p or P_t, we can have different cases described as follows.

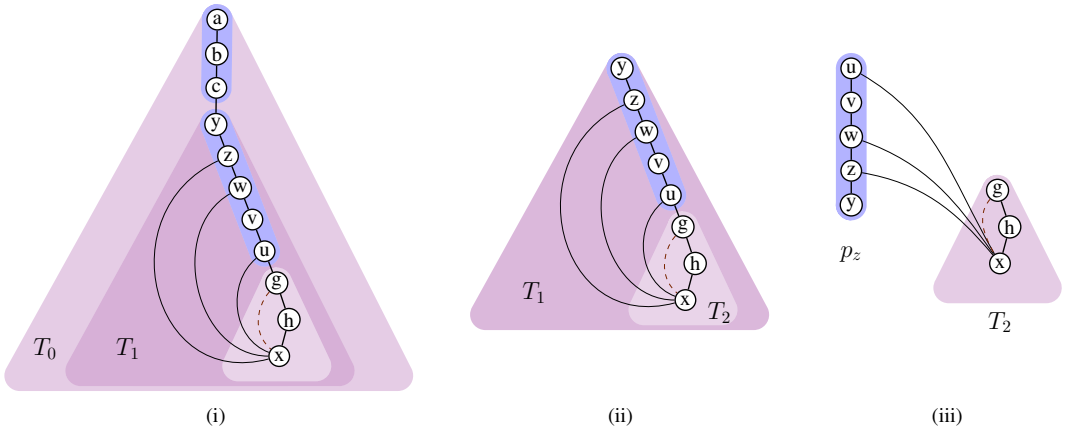

Figure 3.11: (i) Before the beginning of algorithm vertex x belongs to tree $T_0 \in \mathcal{T}$, z is the highest ancestor of x in T_0 such that (x, z) is an edge. (ii) The partitioning changes as the algorithm proceeds, $T_1 (\in \mathcal{T})$ is the tree containing vertex z just before it is attached to T^*. (iii) A path containing vertex z (i.e. p_z) is extracted from T_1 and attached to T^*. If $a(x)$ belongs to T_0, then it is the highest neighbor of x in p_z.

1. Vertex $a(x)$ lies on a path in P_p

For every vertex $v \in V$ and each path $p_0 \in P_p$, we query $\mathcal{D}$ to compute the edge (u, v) where u is closest to $H(p_0)$ on path p_0, and add u to $A(v)$. Note that if $a(x)$ lies on p_0, for $v = x$ the computed vertex u will be same as $a(x)$.

2. Vertex x lies on a path in P_p

   For each $u \in T^*$ and $p_0 \in P_p$, we query $\mathcal{D}$ for an edge (u, y) such that the endpoint y is farthest from $H(p_0)$ on path p_0. We add u to $A(y)$. Now, consider a vertex x on p_0 such that $a(x) = u$. If x is equal to y, then we have added $a(x)$ (i.e. u) to $A(x)$. If x is not equal to y, then we have added $a(x)$ (i.e. u) to $A(y)$ where y is descendant of x in T^*.

3. Vertex x and $a(x)$ lies on same path in P_t

   For every vertex $v \in p_0$ for a path $p_0 \in P_t$, we query $\mathcal{D}$ to compute the edge (u, v) where u is closest to $H(p_0)$ on path p_0, and add u to $A(v)$. Note that for $x = v$, if $a(x)$ also lies on p_0, then u will be same as $a(x)$.

4. Vertex x and $a(x)$ lies on different paths in P_t

   Let x belong to T_0 in the initial disjoint tree partitioning $\mathcal{T} \cup \mathcal{P}$. We claim that $a(x)$ would also belong to same tree T_0. This is because disjoint tree partitioning ensures the absence of edges between two subtrees in $\mathcal{T}$. Let z be the highest ancestor of x in T_0 such that (x, z) is an edge in $G + U$. Let p_z be the path in P_t containing vertex z.

   We now prove that $a(x)$ belongs to p_z. Recall that as the algorithm proceeds, our partitioning $\mathcal{P} \cup \mathcal{T}$ evolves with time. Let T_1 be the tree in $\mathcal{T}$ containing vertex z just before p_z is attached to T^*. Then T_1 is either same as T_0, or a subtree of T_0 (see Figure 3.11 (i)). Also, $a(x)$ must lie in tree T_1, since it cannot be an ancestor of z in T_0. Now, let T_2 be the tree containing x which is obtained on removal of p_z from T_1. Since z is an ancestor of x in T_0, the vertices in T_2 will eventually hang from some descendant of z (not necessarily proper) in T^*. For $a(x)$ to be the highest neighbor of x in T^*, it should be an ancestor of z in T^*, which is only possible if $a(x) \in p_z$.

   Therefore, for each vertex x belonging to a tree T_0 in $\mathcal{T}$, we calculate the highest ancestor z of x in T_0 such that (x, z) is an edge in $G + U$. We compute a list $l(z)$ that consist of all the vertices x whose highest ancestor in T_0 is z. Now, when p_z is added to T^*, we process $l(z)$ as follows. For every $v \in l(z)$, we query $\mathcal{D}$ for an edge (u, v) where u is closest to $H(p_z)$ on path p_z, and add u to $A(v)$. Note that if $a(x)$ also lies in T_0, then u must be same as $a(x)$ (see Figure 3.11 (iii)).

Now, in the first two steps the total time taken is dominated by the number of queries between each path in P_p and the vertices in T, i.e., $|P_p| \times n \times \log^3 n = O(nk \log^4 n)$. In the last two steps the total time taken is dominated by a single query for each vertex in T, i.e., $n \times \log^3 n = O(n \log^3 n)$. Thus, we have the following theorem.

Theorem 3.9. *Given an undirected graph $G(V, E)$ with $|V| = n$ and $|E| = m$, we can maintain a data structure for answering queries of biconnected components and 2 edge connectivity in a dynamic graph which takes $O(\sqrt{mn} \log^{2.5} n)$ update time, $O(1)$ query time and $O(m \log n)$ time for preprocessing.*

3.10 Lower Bounds

We now prove two conditional lower bounds for maintaining a DFS tree under vertex or edge updates.

3.10.1 Vertex Updates

The lower bound for maintaining a DFS tree under vertex updates is based on Strong Exponential Time Hypothesis (SETH) as defined below:

Definition 3.4 (SETH)**.** *For every $\epsilon > 0$, there exists a positive integer k, such that SAT on $k-$CNF formulas on n variables cannot be solved in $\tilde{O}(2^{(1-\epsilon)n})$ time.*

Given an undirected graph G on n vertices and m edges in a dynamic environment (incremental / decremental or fully dynamic) under vertex updates. The status of any vertex can be switched between *active* and *inactive* in an update. The goal of subgraph connectedness is to efficiently answer whether the subgraph induced by active vertices is connected. Abboud and Williams[AW14] proved a conditional lower bound of $\Omega(n)$ per update based on SETH for answering dynamic subgraph connectedness queries. They proved that any algorithm for answering dynamic subgraph connectedness queries using arbitrary polynomial preprocessing time and $O(n^{1-\epsilon})$ amortized update time would essentially refute the SETH conjecture. They also proved that any algorithm for maintaining partially dynamic (incremental/decremental) subgraph connectedness using arbitrary polynomial preprocessing time and $O(n^{1-\epsilon})$ worst case update time would essentially refute the SETH conjecture.

We present a reduction from subgraph connectedness to maintaining DFS tree under vertex updates requiring the algorithm to report whether the number of children of the root in any DFS tree of the subgraph is greater than 1. Thus, we establish the following:

Theorem 3.10. *Given an undirected graph G with n vertices and m edges undergoing vertex updates, an algorithm for maintaining DFS tree that can report the number of children of the root in the DFS tree with preprocessing time $p(m,n)$, update time $u(m,n)$ and query time $q(m,n)$ would imply an algorithm for subgraph connectedness with preprocessing time $p(m+n,n)$, update time $u(m+n,n)$ and query time $q(m+n,n)$.*

Proof. Given the graph G for which we need to query for subgraph connectedness, we make a graph G' as follows. We add all vertices and edges of G to G'. Further, add another vertex r called as *pseudo root* and connect it to all other vertices of G'. Thus, G' has $n+1$ vertices and $m+n$ edges. Now, in any DFS tree T of G' rooted at r, the number of children of r will be equal to the number of components in G. Here subtrees rooted on each child of s represents a component of G. Any change on G can be performed on G' and query for subgraph connectedness in G is equivalent to querying if r has more than 1 child in T. □

Thus, any algorithm for maintaining fully dynamic DFS under vertex updates with arbitrary preprocessing time and $O(n^{1-\epsilon})$ amortized update time would refute SETH. Also, any algorithm for maintaining partially dynamic DFS under vertex updates with arbitrary preprocessing time and $O(n^{1-\epsilon})$ worst case update time would refute SETH.

3.10.2 Edge Updates

We now present a lower bound for maintaining a DFS tree under edge updates that holds for any algorithm which maintains tree edges of the DFS tree explicitly. In the following example we prove that there exists a graph G and a sequence of edge updates U, such that any DFS tree of the graph would require a conversion of $\Omega(n)$ edges from tree edges to back edges and vice-versa after every pair of updates in U.

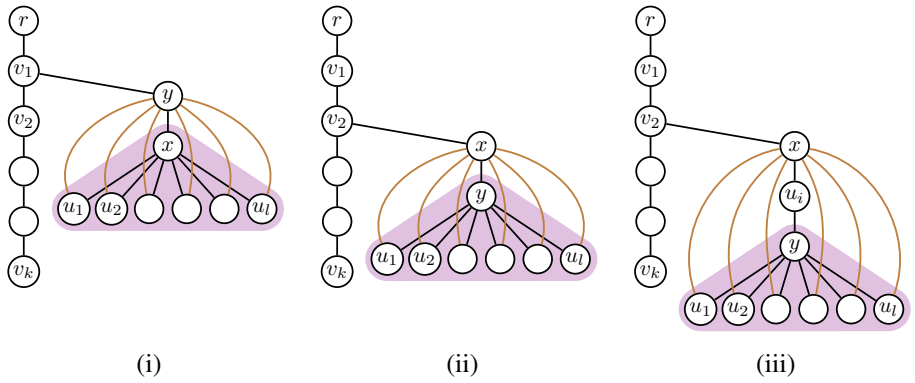

Figure 3.12: Worst Case Example for lower bound on maintaining DFS tree under fully dynamic edge updates.

Consider the following graph for which a DFS tree rooted at r is to be maintained under fully dynamic edge updates. There are $n/2$ vertices u_1,...,u_l that have edges to vertices x and y. The remaining $n/2 - 3$ vertices v_1,..., v_k are connected in form of a line as shown in Figure 3.12. At any point of time one of $v_1, ..., v_k$ (say v_1) is connected to either x or y. The DFS tree for the graph is shown in Figure 3.12 (i). Now, upon insertion of edge (v_i, x) (say $i = 2$) and deletion of edge (v_1, y) the DFS tree will transform to either Figure 3.12 (ii) or Figure 3.12 (iii). Clearly $\Omega(n)$ edges are converted from tree edges to back edges and vice-versa. This can be repeated alternating between x and y ensuring that the new DFS tree requires $\Omega(n)$ after every two edge updates. Further, we repeat this for different v_i's ensuring that the new DFS tree is not exactly the same as some previous DFS tree (thus memorization of the complete tree will not help). Note that the same procedure can be applied to both the possible trees shown in Figure 2.5(ii) and Figure 2.5(iii). Hence any algorithm maintaining tree edges explicitly takes $\Omega(n)$ time to handle such a pair of edge updates.

3.11 Discussion

We have presented a fully dynamic algorithm for maintaining a DFS tree that takes worst case $\tilde{O}(\sqrt{mn})$ update time. This is the first fully dynamic algorithm that achieves $o(m)$ update time. In the fault tolerant setting our algorithm takes $\tilde{O}(nk)$ time to report a DFS tree, where k is the number of vertex or edge failures in the graph. We show the immediate applications of fully dynamic DFS for solving various problems such as dynamic subgraph connectivity, biconnectivity and 2-edge connectivity. We also prove the conditional lower bound of $\Omega(n)$ on maintaining DFS tree under vertex/edge updates.

A natural question is to consider whether the current techniques can be extended to achieve a $\tilde{O}(n)$ update time algorithm for fully dynamic DFS, matching the corresponding lower bound. However, the main problem in being able to do so, is the difficulty in updating the data structure $\mathcal{D}$ whenever the structure of the DFS tree is changed. Using the same data structure for handling multiple updates may not give us an improved bound due to the presence of $\tilde{O}(nk)$ edges among the *super nodes* (elements of disjoint tree partitioning). Another interesting research direction would be to consider similar techniques for directed graphs. Here the primary problem seems to be the traversal of the elements of disjoint tree partitioning, which does not allow *path halving* or traversing the path to the root in a subtree. Moreover, a DFS tree in directed graphs restricts only *anti-cross edges*, allowing *cross edges* from right to left in the DFS tree. This leads to the absence of any bound on the edges among the *super nodes*, like the $\tilde{O}(nk)$ bound in case of undirected graphs.

DFS tree has been extensively used for solving various graph problems in the static setting. Most of these problems are also solved efficiently in the dynamic environment. However, their solutions have not used dynamic DFS tree. Furthermore, solutions to most dynamic graph problems under edge updates requires $o(n)$ update time. However, this is not true for the vertex update variants of these problems. In the light of $\Omega(n)$ lower bound for updating DFS under both edge and vertex updates, it becomes clear that dynamic DFS tree would be more applicable in dynamic graph problems under vertex updates. The applications of our fully dynamic algorithm follows from the fact that it handles vertex updates which was not the case with the existing algorithms for maintaining DFS tree in any dynamic setting. This work is thus an attempt to restore the glory of DFS trees for solving graph problems in the dynamic setting as was the case in the static setting. We believe that our dynamic algorithm for DFS, on its own or after further improvements/modifications, would encourage other researchers to use it in solving various other dynamic graph problems.

Chapter 4

Dynamic DFS in other models of computation

4.1 Introduction

In Section 3.3 we described a simple algorithm for updating the DFS tree of an undirected graph after an edge/vertex update in $\tilde{O}(n)$ time. However, this algorithm is strictly sequential. In this chapter, we present another algorithm achieving similar bounds, that can be easily adopted to more practical models of computation. This algorithm can be used to develop deterministic parallel algorithms for maintaining fully dynamic DFS and fault tolerant DFS of an undirected graph. Our fully dynamic DFS algorithm requires $\tilde{O}(1)$ time per update using m processors. However, if the number of processors are limited to n, our fault tolerant algorithm can report the DFS tree of the graph after any k edge or vertex updates in $\tilde{O}(1)$ time (for constant k). It also provides a fully dynamic DFS algorithm in semi-streaming model requiring $\tilde{O}(1)$ passes per update, and in distributed $\mathcal{CONGEST}(n/D)$ model, requiring $\tilde{O}(D)$ rounds per update, where D is the diameter of the graph. These are the first parallel, semi-streaming and distributed algorithms for maintaining a DFS tree in the dynamic setting.

Our fully dynamic algorithm and fault tolerant algorithm (for constant k), clearly takes optimal time (up to $poly \log n$ factors) for maintaining a DFS tree. Our fault tolerant algorithm (for constant k) is also work optimal (up to $poly \log n$ factors) since a single update can lead to $\Theta(n)$ changes in the DFS tree. Moreover, our result also establishes that maintaining a fully dynamic DFS tree for an undirected graph is in NC (which is still an open problem for DFS tree in the static setting). 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 $\mathcal{C}ONGEST(B)$ 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 [1]. This essentially improves the state of the art for the classes of graphs with $o(n)$ diameter.

[1]For an algorithm maintaining the whole DFS tree at each node, our message size is also optimal. This is because an update of size $O(n)$ (vertex insertion with arbitrary set of edges) has to be propagated throughout the network in the worst case. In $O(D)$ rounds, it can only be propagated using messages of size $\Omega(n/D)$. (see Section 4.10.2 for details).

4.2 Overview

We now describe a brief overview of our result. In Section 3.3 we proved that updating a DFS tree after any update in the graph is equivalent to *rerooting* disjoint subtrees of the DFS tree. We 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 at 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 that result, we used a property of the DFS tree, called *components* property, to find the edge (x, y) efficiently, using a data structure (referred as $\mathcal{D}_0$ in this chapter). However, as evident from the discussion above, that 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. In this chapter we develop an algorithm that performs this rerooting efficiently in parallel.

Our new 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 Section 3.3 every component of the unvisited graph is of type $C1$. We define three types of traversals, namely, *path halving*, *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 new algorithm ensures that each vertex is queried by $\mathcal{D}$ only $\tilde{O}(1)$ times (unlike one in Section 3.3), our new data structure $\mathcal{D}$ is much simpler than $\mathcal{D}_0$.

Furthermore, both these algorithms 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 the one in Section 3.3), 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 $\mathcal{D}$.

4.3 Preliminaries

For our distributed algorithm, we use the synchronous $\mathcal{CONGEST}(B)$ model [Pel00]. For the dynamic setting, Henzinger et al. [HKN13] presented a model that has a *preprocessing* stage followed by an alternating sequence of non-overlapping stages for *update* and *recovery* (see Section 4.10.2 for details). We use this model with an additional constraint of space restriction of $O(n)$ size at each node. In the absence of 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.

We shall now define some queries that are performed by our algorithm on the data structure $\mathcal{D}$ (similar to the queries on $\mathcal{D}_0$ in Chapter 3). 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.

Recall the properties of a DFS tree in undirected graphs described in Section 3.3. We also described how after any update in the graph, the DFS tree of the updated graph can be evaluated by *rerooting* disjoint subtrees of the current DFS tree T, using the *components property*. This reduction will henceforth be referred as the *reduction* algorithm. Since each of these disjoint subtrees can be rerooted independent of each other, it can be performed in parallel to each other. However, in order to perform the *reduction* algorithm efficiently in parallel, we require a structure to answer the following queries efficiently in parallel (see Section 3.3). (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 as described in Section 4.9.1. Thus, we have the following theorem.

Theorem 4.1. *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 $\mathcal{D}$.

4.4 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 4.3). Further, 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:

$C1$: A single subtree τ_c of the DFS tree T.

$C2$: 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.

4.5 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 Procedure Reroot-DFS in Section 4.8 for the pseudo-code of the main 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 $v_l = LCA(r_c, v_H)$.

4.5.1 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 $path(r_c, v_H)$, adding it to T^* (see Figure 4.1 (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. Refer to Procedure DisInt-DFS in Section 4.8 for the pseudo code of the traversal.

Now, 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 4.3). Thus, every component c^* can be identified and moved to next phase. The pseudocode for this procedure of identifying the components and their corresponding roots, and moving them to the next stage is described in Procedure Process-Components.

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).*

4.5.2 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 4.1 (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 $p_{c'}$

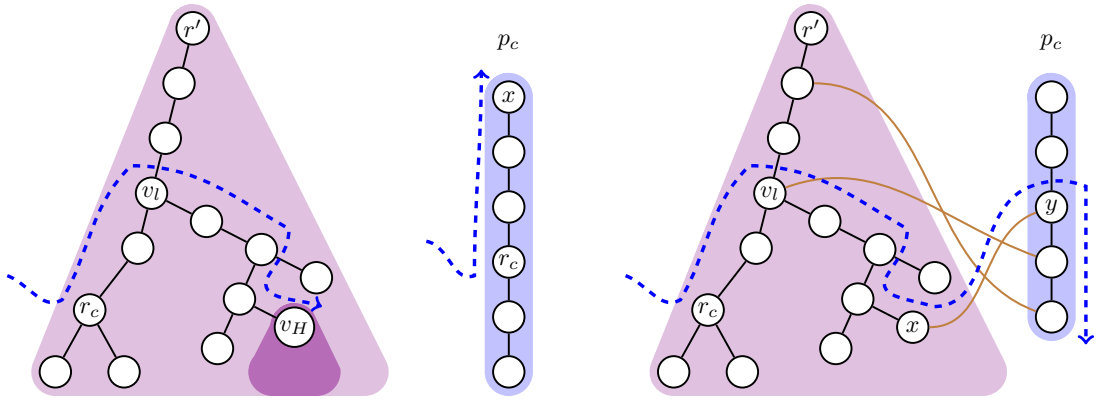

Figure 4.1: The three *simpler* traversals (shown using blue dotted lines), (a) Disintegrating traversal, (b) Path Halving and (c) Disconnecting traversal.

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^*. Refer to Procedure Path-Halving-DFS in Section 4.8 for the pseudo code.

4.5.3 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 4.1 (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 4.5.1. 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 (see Procedure DisCon-DFS in Section 4.8 for pseudo code).

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, i.e. violate the phase/stage constraints*. Hence, in such a case we would process the component using heavy subtree traversal described as follows.*

4.5.4 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, namely l, p or r traversal (see Figure 4.2). Our algorithm checks each scenario in turn for its applicability to τ, eventually choosing a scenario if it ensures that it can be followed by a *simpler* traversal described earlier, to move each component to the next phase or the next stage. We shall also show 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. The three conditions of this lemma ensures that the traversal can indeed be applied based on the discussion above. Here, $\mathcal{A}_1$ ensures the invariant of component types, $\mathcal{A}_2$ ensures the phase and stage constraints, and $\mathcal{A}_3$ ensures that the next traversal in not heavy subtree traversal (i.e., the new root is, either not from a heavy subtree, or is the root of the heavy subtree (may be a tree edge), or is from $T(v_H)$).

Lemma 4.5.1 (Applicability Lemma)**.** *If there exists a path p^* starting from r_c 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 p^* satisfies the following three conditions*

- $\mathcal{A}_1$*: Traversal of p^* produces components of type $C1$ or $C2$ only,*
- $\mathcal{A}_2$*: If the subtree $T(v_H)$ is in a component c' of type $C2$, then $p_{c'} = p_c$.*
- $\mathcal{A}_3$*: The lowest edge on p^* from the component containing p_c is* ***not*** *the following edge: an edge from a heavy subtree (the subtree containing $T(v_H)$), which is a back edge with its endpoint outside $T(v_H)$.*

Proof. Consider any traversal satisfying the above criteria, which forms components of type $C1$ and $C2$ only (by $\mathcal{A}_1$). For each such component c', we find the lowest edge e' from c' to the traversed path, giving the new root $r_{c'}$. Every component which does not contain p_c or $T(v_H)$ can be directly moved to the next phase with root $r_{c'}$, because the remaining subtrees of τ (not containing $T(v_H)$) cannot be heavy. In case the component containing $T(v_H)$ is of type $C1$ it can be moved to the last stage of the phase. In case the component c' containing p_c does not contain $T(v_H)$, we have $r_{c'} \in p_c$ or $r_{c'} \in \tau'$ (a non-heavy subtree of τ), moving c' to the next stage after performing *path halving* or *disconnecting* traversal of τ' respectively. Due to $\mathcal{A}_2$, this only leaves the component c' of type $C2$ having both p_c and a subtree $T(v_h) \in \mathcal{T}_{c'}$ which contains $T(v_H)$.

$\mathcal{A}_3$ prevents e' from satisfying the following three criteria simultaneously: (i) $r_{c'} \in T(v_h)$ (heavy subtree), (ii) e' is a back edge, and (iii) $r_{c'} \notin T(v_H)$. In case any one of these three criteria for e' is false, the traversal of p^* can be followed by a *simpler* traversal as follows. If $r_{c'} \notin T(v_h)$, it either belongs to a light subtree or p_c ensuring *disconnecting* traversal or *path halving* respectively. Otherwise when $r_{c'} \in T(v_h)$, being a tree edge or

having $r_{c'} \in T(v_H)$ ensures *disintegrating* traversal (as $r_{c'} = root(T(v_h))$) or *disconnecting* traversal respectively. □

Remark. *Applicability lemma is employed when p^* does not pass 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 resulting subtrees of the new component containing p_c would only have light subtrees.*

Overview

We now present a brief overview of the heavy subtree traversal using the applicability lemma. Essentially we would attempt to perform a traversal that implicitly satisfies the $\mathcal{A}_1$ and $\mathcal{A}_2$, and we will verify whether it satisfies $\mathcal{A}_3$. After any traversal let the subtree hanging from the traversed path that contains v_H be $T(v_h)$. Further, let any subtree hanging from the traversed path be called an *eligible* subtree, if it has an edge to p_c.

First attempt the simplest traversal for the subtree that goes all the way to the root r' from r_c. This shall be later referred as l traversal (see Figure 4.2). Clearly it satisfies $\mathcal{A}_1$ and $\mathcal{A}_2$. The traversal will not be applicable if it fails $\mathcal{A}_3$. To verify $\mathcal{A}_3$ we find the lowest edge (x_1, y_1) on the traversed path (highest on $path(r_c, r')$) from the eligible subtrees, where y_1 is on the traversed path. Thus, $\mathcal{A}_3$ will not be satisfied only if this lowest edge is from $T(v_h)$, i.e., $x_1 \in T(v_h)$.

Second attempt would be to use the previous failed condition to our advantage and perform a traversal using this back edge (x_1, y_1). Since it was the highest edge, no eligible subtree can be connected to $path(par(y_1), r')$. Hence we perform a traversal following the tree path to x_1, followed by (x_1, y_1) and then downwards to $par(v_l)$. This leaves $path(par(y_1), r')$ untraversed, which is not connected to any eligible subtree. This shall be called as a p traversal (see Figure 4.2 (b) assuming $(x_p, y_p) = (x_1, y_1)$). Clearly, such a traversal satisfies $\mathcal{A}_1$ and $\mathcal{A}_2$. To verify $\mathcal{A}_3$ we find lowest edge (x_2, y_2) on traversed path (having suffix as $path(y_1, par(v_l))$) from the eligible subtrees, where y_2 is on the traversed path. In this case, $\mathcal{A}_3$ will not be satisfied only if this lowest edge is from $T(v_h)$ on $path(y_1, par(v_l))$, i.e., $x_2 \in T(v_h)$.

Third attempt would be to use even the second failed condition to our advantage and perform a traversal using this back edge (x_2, y_2). Since it was lowest edge, no eligible subtree can be connected to $path(y_1, par(v_l))$. Hence we perform a traversal following the tree path to x_2, followed by (x_2, y_2) and then upwards to r_c. This shall be called as a r traversal (see Figure 4.2 (c) assuming $(x_r, y_r) = (x_2, y_2)$). Clearly it satisfies $\mathcal{A}_1$ and $\mathcal{A}_2$ (except for a *minor* case described later). To verify $\mathcal{A}_3$ we find lowest edge (x_3, y_3) on traversed path (highest on the suffix as $path(y_2, r')$) from the eligible subtrees, where y_3 is on the traversed path. Again, $\mathcal{A}_3$ will not be satisfied if this lowest edge is from $T(v_h)$ on $path(y_2, r')$, i.e., $x_3 \in T(v_h)$.

Notice that we can also use this new edge (x_3, y_3) (instead of (x_1, y_1)) for p traversal described earlier and it would still satisfy $\mathcal{A}_1$ and $\mathcal{A}_2$ (except for a *minor* case described later). Further, we would also be making progress as whenever the attempt fails, the computed edge is from the new $T(v_h)$. And this edge would be from smaller heavy subtree because the corresponding x will get closer to the inner core $T(v_H)$, i.e. has lower $LCA(x, v_H)$ ($LCA(x_1, v_H) > LCA(x_2, v_H) > LCA(x_3, v_H)$). Thus, we can alternate between attempting p traversal and r traversal using the newly computed edge, making

progress to eventually reach the point when $\mathcal{A}_3$ is satisfied. However, this progress can take up to $O(n)$ such attempts, and is the reason for *sequentiality* of the procedure. In our main algorithm we show how to perform this procedure using $O(1)$ attempts.

Further, note that both (x_1, y_1) and (x_3, y_3) are highest edges on $path(r_c, r')$ from the eligible subtrees after l and r traversal respectively. Consider $T(v_h)$ after r traversal, for any $x \in T(v_h)$ we have $LCA(x, v_H)$ lower than $LCA(x_2, v_H)$. Since $LCA(x_1, v_H)$ is higher than $LCA(x_2, v_H)$, we have $x_1 \notin T(v_h)$, i.e., (x_1, y_1) does not belong to a heavy subtree after r traversal. Hence, if (x_3, y_3) was same as (x_1, y_1), $\mathcal{A}_3$ would have been satisfied by the r traversal. Thus, it is sufficient for *applicability of r traversal* that $(x_3, y_3) = (x_1, y_1)$. We now describe the condition for (x_1, y_1) which ensures $(x_3, y_3) = (x_1, y_1)$. Since $x_1 \notin T(v_h)$, either $x_1 \in path(v_l, v_H)$ or x_1 belongs to a light subtree τ_1 hanging from $path(v_l, v_H)$. Further, y_1 would always be at least as high as y_3, because it is computed from a less restricted set of trees. Thus, (x_3, y_3) is same as (x_1, y_1) except when (x_1, y_1) no longer belongs to an eligible subtree after the r traversal. This is possible only if τ_1 is not eligible or x_1 has been traversed by r traversal. Note that since $LCA(x_2, v_H)$ is lower than $LCA(x_1, v_H)$, x_1 can only be traversed by r traversal if $x_1 \in path(v_l, v_H)$. Thus, to ensure $(x_1, y_1) = (x_3, y_3)$ and hence *applicability of r traversal*, we require $x_1 \in \tau_1$ where τ_1 is an eligible subtree hanging from $path(v_l, v_H)$.

Thus, we select the edge for p traversal (instead of simply using (x_1, y_1)) in such a way that both *non-applicability* of r traversal and *sequentiality* of the procedure can be avoided. This can be achieved by a careful selection of two edges (x_d, y_d) and (x_p, y_p), such that if (x_p, y_p) is used for p traversal (instead of (x_1, y_1)), then after the r traversal the edge (x_3, y_3) is same as (x_d, y_d) (instead of (x_1, y_1)), which would result in $\mathcal{A}_3$ being satisfied for the r traversal.

Remark. *There is a* minor case *in which the* $\mathcal{A}_1$ *and* $\mathcal{A}_2$ *will not be satisfied in the r traversal and subsequent p and r traversals described above. Handling this* minor case *leads to a* special case *in which even* (x_d, y_d) *may not ensure that* $\mathcal{A}_3$ *is satisfied after the r traversal. In such a case, we simply augment the r traversal with another simple traversal eventually satisfying* $\mathcal{A}_3$*.*

Definitions

We now briefly describe the three scenarios, namely, l, p and r traversals and define a few notations related to them (shown in Figure 4.2). 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 (similar to $T(v_h)$ described earlier). Recall that 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 and p_c, 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 $x_d \in \tau_d \neq T(v_R)$.

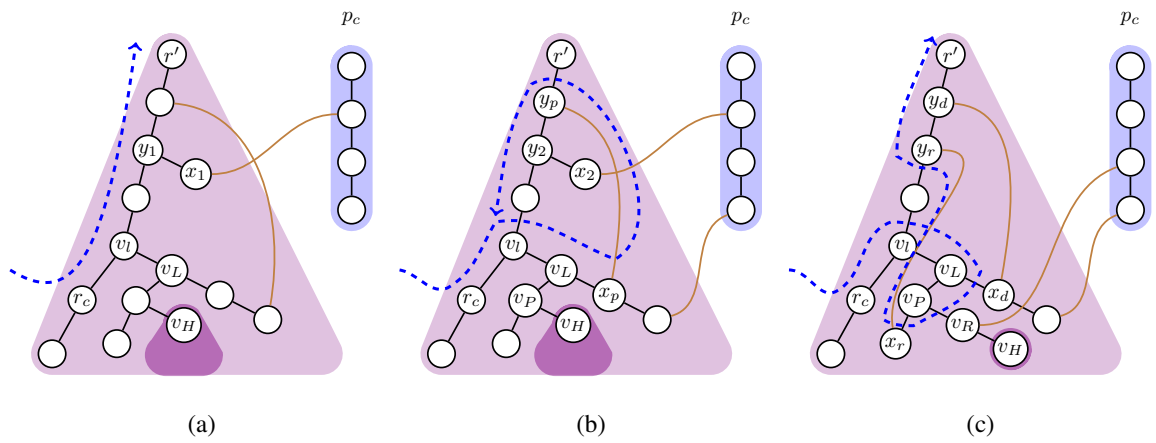

Figure 4.2: The three scenarios for Heavy Subtree Traversal (shown using blue dotted lines), (a) l traversal, (b) p traversal, and (c) r traversal.

Scenario 1: l traversal

Consider the traversal of $p_L^* = path(r_c, r')$. Since, this traversal does not create a new non-traversed path, $\mathcal{A}_1$ and $\mathcal{A}_2$ are implicitly satisfied. Let (x_1, y_1) be the lowest edge on p_L^* (highest edge on $path(r_c, r')$) from p_c and the *eligible* subtrees, where $y_1 \in p_L^*$. In case (x_1, y_1) satisfies $\mathcal{A}_3$, we perform the traversal otherwise move to the next scenario. Consider Figure 4.2 (a), clearly l traversal is applicable as (x_1, y_1) is not from a heavy subtree. However, in Figure 4.2 (b) and Figure 4.2 (c), (x_1, y_1) would be (x_p, y_p) and (x_d, y_d) respectively, both of which does not satisfy $\mathcal{A}_3$.

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

Scenario 2: p traversal

Consider the traversal of $p_P^* = path(r_c, x_p) \cup (x_p, y_p) \cup path(y_p, par(v_l))$. 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. We now describe how such a choice of (x_p, y_p) and (x_d, y_d) can be made. 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.

Choice of (x_p, y_p) and (x_d, y_d)

We find the highest edge on $path(r_c, r')$ from the *eligible* subtrees hanging from p_L^* except $T(v_L)$, and the subtrees hanging from $path(v_L, v_H)$ having a back edge to p_c. This edge is stored as (x_d, y_d), where $y_d \in path(r_c, r')$, and shall be used in Scenario 3. The intuition behind such a computation of (x_d, y_d) can be derived from the condition for *applicability of* r *traversal* (see overview). The corresponding subtree (among the queried subtrees), to which x_d belongs is τ_d. Note that (x_d, y_d) may not necessarily exist. Next, we find the edge (x_p, y_p) as follows. Consider all the back edges (x_i, y_i) having $x_i \in T(v_L)$ and $y_i \in path(y_d, r')$ (or $y_i \in path(r_c, r')$ if (x_d, y_d) does not exist). The tree path $path(y_i, x_i)$

deviates from $path(v_L, v_H)$ on the vertex $LCA(x_i, v_H)$. We choose (x_p, y_p) as the edge (x_i, y_i) whose corresponding tree path $path(y_i, x_i)$ deviates lowest from $path(v_L, v_H)$, i.e., having the lowest $LCA(x_i, v_H)$. The intuition behind such a computation of (x_p, y_p) can described as follows. *Firstly*, $y_i \in path(y_d, r')$ ensures that after p traversal, no eligible subtree (including τ_d) is connected to remaining part of $path(r_c, r')$ (by definition of (x_d, y_d)) violating $\mathcal{A}_1$. *Secondly*, choosing an edge with lowest $LCA(x_i, v_H)$ avoids the *sequentiality* of our procedure (see overview), This procedure to compute (x_p, y_p) is crucial to prove the desired properties of (x_p, y_p) and (x_d, y_d) (later in Scenario 3). Additionally, the existence of back edge (x_1, y_1) (see remark of l traversal) implies the following property of the computed edges (x_p, y_p).

Lemma 4.5.2. *The edge (x_p, y_p), which is a back edge, always exists and when used for p traversal satisfies $\mathcal{A}_1$ and $\mathcal{A}_2$.*

Proof. Recall that (x_1, y_1) is the highest edge on $path(r_c, r')$ from the *eligible* subtrees hanging from the p^*_L , whereas (x_d, y_d) is the highest edge on $path(r_c, r')$ from a more restricted set of subtrees. Thus, y_1 is at least as high as y_d on $path(r_c, r')$. Now, since (x_1, y_1) is a back edge with $x_1 \in T(v_L)$ (see remark in Scenario 1), (x_1, y_1) is also a valid edge for (x_p, y_p) ensuring its existence of p edge. Further, (x_p, y_p) is also a back edge because (x_1, y_1) is a back edge and $LCA(x_p, v_H)$ is at most as high as $LCA(x_1, v_H)$ ensuring $x_p \neq v_L$.

Now, consider the p traversal using (x_p, y_p), which produces an untraversed path, say $p' = path(par(y_p), r')$. To prove that this traversal produces only components of type $C1$ and $C2$ (the condition $\mathcal{A}_1$), we only need to prove that any *eligible* subtree (subtree hanging from p^*_P with an edge to p_c) is not connected to p'. This is because p' itself is not connected directly to p_c by an edge (as $x_1 \notin p_c$). We first prove this property for the subtrees queried for finding (x_d, y_d). Since y_p is at least as high as y_d on $path(r_c, r')$, any such subtree will not be connected to p'. Now, we are left to prove this property for the remaining subtrees of $T(v_L)$ hanging from p^*_P with an edge to p_c. The only such subtree is $T(v_P)$, the subtree hanging from p^*_P which contains $T(v_H)$ (satisfying $\mathcal{A}_2$). Since, among all the edges (x, y) from $T(v_L)$ to $path(y_d, r')$, x_p is the vertex with lowest $LCA(x, v_H)$, the subtree $T(v_P)$ is not connected to p'. This is because for any such edge (x, y), where $x \in T(v_P)$, would have $LCA(x, v_H)$ lower than v_P, which is clearly lower than $LCA(x_p, v_H)$ on $path(v_L, v_H)$. □

Lemma 4.5.2 ensures that our traversal can follow p^*_P satisfying $\mathcal{A}_1$ and $\mathcal{A}_2$. To verify $\mathcal{A}_3$ we find the new root for the component having path p_c as follows. Let (x_2, y_2) be the lowest edge on p^*_P from p_c and the *eligible* subtrees hanging from p^*_P, where $y_2 \in p^*_P$. In case (x_2, y_2) satisfies $\mathcal{A}_3$, we perform the traversal otherwise move to the next scenario. Consider Figure 4.2 (b), clearly p traversal is applicable as (x_2, y_2) is not from a heavy subtree. However, in Figure 4.2 (c) we have (x_p, y_p) same as (x_d, y_d), and hence (x_2, y_2) is (x_r, y_r) which clearly does not satisfy $\mathcal{A}_3$.

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

Scenario 3: r traversal

Consider the traversal of $p^*_R = path(r_c, x_r) \cup (x_r, y_r) \cup path(y_r, r')$. We choose (x_2, y_2) as (x_r, y_r). The r traversal using this edge creates an untraversed path on $path(r_c, r')$, i.e.,

$path(par(v_l), y_2)\backslash\{y_2\}$. However, such a traversal would not necessarily satisfy $\mathcal{A}_1$ and $\mathcal{A}_2$ (recall the *minor case* mentioned in the overview) as follows. While computing (x_2, y_2), some portion of τ_p (if exists) could have been a part of p^*_P. Hence, if we perform r traversal using (x_2, y_2) as (x_r, y_r), it is possible that τ_p (now untraversed by p^*_R) is connected to this untraversed $path(par(v_l), y_2)\backslash\{y_2\}$ in addition to p_c. Thus, we would have a component which is not of type C_1 or C_2 violating $\mathcal{A}_1$. We avoid this problem as follows. Let (x'_2, y'_2) be the lowest edge from τ_p to $path(r_c, r')$, where $y'_2 \in path(r_c, r')$. Thus, if y'_2 is lower than y_r on $path(r_c, r')$, 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 4.5.3. *The edge (x_r, y_r), which is a back edge, always exists and when used for r traversal satisfies $\mathcal{A}_1$ and $\mathcal{A}_2$.*

Proof. Existence of (x_2, y_2) clearly implies the existence of (x_r, y_r). Further, (x_r, y_r) is a back edge since both choices for it are back edges, i.e., (x_2, y_2) (see remark in Scenario 2) and (x'_2, y'_2) (as $root(\tau_p) \neq v_L$).

Now, consider the r traversal using (x_r, y_r), which produces an untraversed path, say $p' = path(par(v_l), y_r)\backslash\{y_r\}$. To prove that this traversal produces only components of type $C1$ and $C2$ (hence satisfies $\mathcal{A}_1$), we only need to prove that any *eligible* subtree (subtree hanging from p^*_R with an edge to p_c) is not connected to p'. This is because p' itself is not connected directly to p_c by an edge (as $x_2 \notin p_c$). Now, the only such subtrees not queried while computing (x_2, y_2) is τ_p, which is queried while computing (x'_2, y'_2). Hence for either choice for (x_r, y_r) ((x_2, y_2) or (x'_2, y'_2)), no *eligible* subtree will be connected to p' (ensuring $\mathcal{A}_2$). □

Thus, Lemma 4.5.3 ensures that our traversal can follow p^*_R as shown in Figure 4.2 (c). To verify $\mathcal{A}_3$ we find the new root for the component having path p_c as follows. Let (x_3, y_3) be the lowest edge on p^*_R from p_c and the *eligible* subtrees hanging from p^*_R, where $y_3 \in p^*_R$. In case this edge satisfies $\mathcal{A}_3$, we perform this traversal otherwise it can be proved to be a *special case* which can be handled by a modified r traversal described in the next section. Refer to Procedure Heavy-DFS in Section 4.8 for the pseudo code of this traversal. Consider Figure 4.2 (c), clearly r traversal is applicable as $(x_3, y_3) = (x_d, y_d)$ is not from a heavy subtree.

Remark. *This scenario is not applicable only if (x_3, y_3) is a back edge with $x_3 \in T(v_R) \setminus T(v_H)$.*

The following lemma describes when (x_3, y_3) satisfies the *applicability* lemma. Our aim was to compute (x_d, y_d) such that it can be used after the r traversal to satisfy $\mathcal{A}_3$ (i.e. $(x_3, y_3) = (x_d, y_d)$), to make the r traversal eligible (see overview). However, because of the *minor case* described earlier this may not always be true. We now present the conditions which necessarily makes it true.

Lemma 4.5.4. *The r traversal using the edge (x_r, y_r) is applicable with $(x_3, y_3) = (x_d, y_d)$, if either $\tau_d \neq \tau_p$ or $(x_r, y_r) \neq (x'_2, y'_2)$.*

Proof. We shall prove this in two steps. First, we will prove that no edge from an eligible subtree on p^*_R would be lower than (x_d, y_d). Thereafter, we would prove why $\mathcal{A}_3$ would be satisfied making r traversal eligible if $(x_3, y_3) = (x_d, y_d)$ holds and the conditions for the same.

Recall the computation of (x_p, y_p) and (x_d, y_d). Since (x_p, y_p) is the edge from $T(v_L)$ to $path(y_d, r')$ whose tree path deviates lowest from $path(v_L, v_H)$, the subtree $T(v_P)$ does not have an edge to $path(y_d, r')$. Now, $T(v_R)$ is a subtree of $T(v_P)$ as $x_r \in T(v_P) \cup \tau_p$ (recall (x_2, y_2) and (x'_2, y'_2)), and hence $T(v_R)$ cannot have an edge on $path(r_c, r')$ higher than (x_d, y_d). Also, the remaining eligible subtrees queried while computing (x_3, y_3) were also queried while computing (x_d, y_d). Hence, no edge from an eligible subtree on p^*_R would be lower than (x_d, y_d).

Clearly if $(x_3, y_3) = (x_d, y_d)$, $\mathcal{A}_3$ is satisfied as τ_d is a light subtree. Hence after the r traversal, $\mathcal{A}_3$ is violated or $(x_3, y_3) \neq (x_d, y_d)$ only if (x_d, y_d) is no longer an edge from an eligible subtree. Note that this is possible if p^*_R passes through τ_d, and either traverses x_d or disconnects it from p_c. Now, if $(x_r, y_r) = (x_2, y_2)$ p^*_R cannot pass through τ_d. This is because $T(v_P)$ and τ_d are disjoint as $LCA(x_p, v_H)$ is at least as low as $LCA(x_d, v_H)$ (by definition of (x_p, y_p)). Thus, it is only possible when $(x_r, y_r) = (x'_2, y'_2)$ because τ_p can be same as τ_d. Hence, $(x_3, y_3) \neq (x_d, y_d)$ only if both $\tau_p = \tau_d$ and $(x_r, y_r) = (x'_2, y'_2)$. □

We now present an overview of how the *special case* can be handled. Using Lemma 4.5.4, the special case is when r traversal is not applicable and hence $(x_r, y_r) = (x'_2, y'_2)$ and $\tau_p = \tau_d$. 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 described as follows.

Special case of heavy subtree traversal

In this section (x_p, y_p) and (x_r, y_r) correspond to the back edges used in p and r traversals described earlier, whereas $(x_{r'}, y_{r'})$ corresponds to the back edge used in modified r' traversal described below. We now recall the conditions leading to the *special* case, and describe its implications on (x_p, y_p), (x_r, y_r) and (x_d, y_d).

Lemma 4.5.5. *The conditions of the special case are (i) $x_3 \in T(v_R)$, (ii) $x_p \in \tau_d$, and (iii) $(x_r, y_r) = (x'_2, y'_2)$. Following are the properties of (x_p, y_p) and (x_d, y_d) in this case.*

1. *$y_d = y_p$, y_3 is lower than y_d on $path(r_c, r')$, and y_r is lower than y_2 on $path(r_c, r')$.*
2. *No subtree of τ_d hanging from $path(root(\tau_d), x_p)$, with an edge to p_c, has an edge lower than y_2 on $path(r_c, r')$.*
3. *No subtree of τ_d hanging from $path(root(\tau_d), x_r)$, with an edge to p_c, has an edge higher than y_3 on $path(r_c, r')$.*

Proof. Recall the choice of (x_d, y_d) and (x_p, y_p) (see Section 4.5.4), it was the highest edge from τ_d on $path(r_c, r')$ and (x_p, y_p) was computed such that $y_p \in path(y_d, r')$. Hence, if $x_p \in \tau_d$ we necessarily have $y_d = y_p$. Also, y_3 is strictly lower than y_p (or y_d) else having a lower $LCA(x_3, v_H)$ than $LCA(x_p, v_H)$, (x_3, y_3) would have been selected as (x_p, y_p) earlier. Finally, y_2 is strictly higher than y_r otherwise (x_2, y_2) would have been selected as (x_r, y_r) earlier. Second property holds since (x_2, y_2) (where $x_2 \in T(v_P)$) was the lowest edge on $path(r_c, r')$ from the eligible subtrees after p traversal. Third property holds since (x_3, y_3) (where $x_3 \in T(v_R)$) was the highest edge on $path(r_c, r')$ from the eligible subtrees after r traversal. □

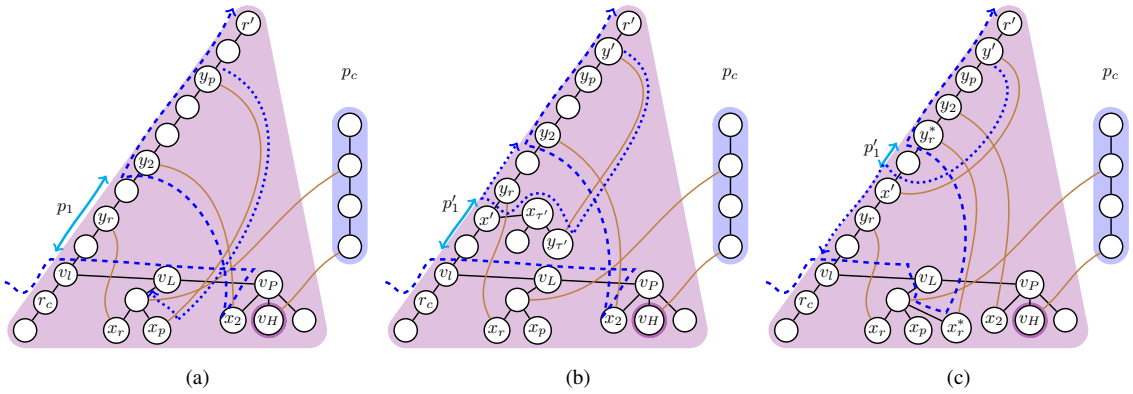

Figure 4.3: The three traversals for *special case* of Heavy Subtree Traversal (shown using blue dotted lines) followed by a modified r' traversal (shown using blue dashed lines). (a) Root traversal of τ_d, (b) Upward cover traversal of p_1 through τ', and (c) Downward cover traversal of p_1 using a direct edge (x', y').

To handle this special case, we revisit the scenario corresponding to r traversal ignoring the *eligible* subtree τ_d. Hence, we choose $(x_{r'}, y_{r'}) = (x_2, y_2)$ despite having a lower edge (x_r, y_r). Thus, after r' traversal we have a subtree τ_d connected to two paths, p_c and the unvisited part of $path(r_c, r')$, which violates $\mathcal{A}_1$. Now, based on the lowest edge from component containing p_c on the traversed path $p^*_{R'}$, we append simple traversals to this modified r' traversal in order to satisfy $\mathcal{A}_1$. We shall shortly see that in these traversals the conditions $\mathcal{A}_2$ and $\mathcal{A}_3$ are implicitly true.

Modified r' traversal

Consider a modified r' traversal using $(x_{r'}, y_{r'}) = (x_2, y_2)$ of the path $p^*_{R'} = path(r_c, x_2) \cup (x_2, y_2) \cup path(y_2, r')$ (see Figure 4.3 (a)). This leaves an untraversed part of $path(r_c, r')$, i.e., $path(par(v_l), y_2) \backslash \{y_2\}$ (say p_1). Using Lemma 4.5.5 we know that y_2 is strictly higher than y_r ensuring $y_r \in p_1$. Now, the computation of (x_2, y_2) ensures that no *eligible* subtree except for τ_d is connected to path p_1 as well as the lowest edge on $p^*_{R'}$ from an *eligible* subtree is from τ_d, i.e., (x_d, y_d). This implicitly satisfies $\mathcal{A}_2$ and $\mathcal{A}_3$, if the appended traversal does not traverse any part of $T(v_{R'})$ or p_c. However, since τ_d is connected to two paths p_1 and p_c (violating $\mathcal{A}_1$), we need to append $p^*_{R'}$ with another traversal which disconnects the unvisited part of τ_d from the unvisited part of p_1.

Now, the only *eligible* subtree having an edge to p_1 is τ_d. Still the lowest edge on $p^*_{R'}$ from the component having p_c may not be from τ_d. This is because p_1 and subtrees connected to it are also in the component containing p_c (since τ_d has an edge on p_1). Thus, depending on the lowest edge on $p^*_{R'}$ from the component containing p_c, say (x, y) with $y \in p^*_{R'}$, we have three cases (see Figure 4.3). If $x \in \tau_d$, we simply perform a *root traversal* of τ_d exploiting property 2 of Lemma 4.5.5 to disconnect p_c and p_1. On the other hand if $x \in p_1$ or some subtree connected to p_1, we shall perform a *cover traversal* of p_1, which visits all vertices on p_1 that are connected to τ_d. As a result, the unvisited part of p_1 is disconnected from τ_d. Since y_r is the lowest edge from τ_d on p_1, in case x is lower than y_r on p_1, we simply traverse upwards covering y_d and other endpoints of edges from τ_d incident on p_1. Else we traverse downwards after making sure that no endpoints of edges

from τ_d that incident on p_1 are above x.

Root Traversal of τ_d

In this case the lowest edge on $p^*_{R'}$ from component containing p_c is from τ_d, i.e., (x_d, y_d). The traversal of $p^*_{R'}$ is followed by the traversal of $p_{R'} = (y_p, x_p) \cup (x_p, root(\tau_d))$ as shown in Figure 4.3 (a). Hence, using property 2 in Lemma 4.5.5, no subtree of τ_d hanging from $p_{R'}$ is connected to p_c (satisfying $\mathcal{A}_1$). As described earlier, since $T(v_{R'})$ does not have an edge to τ_d, after the traversal of $p^*_{R'} \cup p_{R'}$, the component having p_c would have the new root in τ_d or p_c (satisfying $\mathcal{A}_3$).

Cover traversal of p_1

In this case the lowest edge from component containing p_c on $p^*_{R'}$ is from p_1, say (x', y'), or some subtree τ', say $(y_{\tau'}, y')$, which is connected to p_1, where $y' \in p^*_{R'}$. If connected through τ', we choose the highest edge $(x', x_{\tau'})$ from τ' on p_1, with $x' \in p_1$. If x' is lower than y_r, we perform the *upward traversal* towards y_2. Otherwise we perform the *downward traversal* towards v_l. In case of upward traversal, when connected through τ', we update $(x', x_{\tau'})$ to be the lowest edge from τ' on p_1, which still maintains x' to be lower than y_r. Note that this choice of $(x', x_{\tau'})$ ensures that $path(y_{\tau'}, x_{\tau'})$ is a *disconnecting* traversal of τ' from p_1 in both upward and downward traversals. We also define this path from y' to x' as $p_{\tau'}$, i.e., when connected through τ' (see Figure 4.3 (b)), we have $p_{\tau'} = (y', y_{\tau'}) \cup path(y_{\tau'}, x_{\tau'}) \cup (x_{\tau'}, x')$. Otherwise, in case of direct edge when $x' \in p_1$ (see Figure 4.3 (c)), we have $p_{\tau'} = (y', x')$.

1. **Upward traversal on p_1**
   In case x' is lower than y_r, the traversal of $p^*_{R'}$ is followed by the traversal of $p_{R'} = p_{\tau'} \cup path(x', y_2) \backslash \{y_2\}$ (see Figure 4.3 (b)). Since $p_{R'}$ is a *disconnecting* traversal of τ' from p_1, the unvisited part of p_1, say p'_1, is not connected to the unvisited part of τ'. Also, p'_1 is not connected to τ_d and hence the component containing p_c as $y_r \notin p'_1$. Since the unvisited part of τ' is also not connected to p_c, $\mathcal{A}_1$ is satisfied. As described earlier, the component having p_c will have the new root in τ_d, being the only part of the component connected to $p_{R'}$. Since τ_d is not a heavy subtree, $\mathcal{A}_3$ is also satisfied.

2. **Downward traversal on p_1**
   In case x' is higher than y_r and we follow the traversal downwards, $path(x', y_2) \backslash \{y_2\}$ might still have edges from τ_d. Hence, we modify the traversal of $p^*_{R'}$ as follows. Let the lowest edge on $path(x', y_2) \setminus \{x', y_2\}$ from τ_d be (x^*_r, y^*_r), where $y^*_r \in p_1$. In case (x^*_r, y^*_r) doesn't exist, we simply choose $(x^*_r, y^*_r) = (x_2, y_2)$. We now perform a modified r'' traversal using $(x_{r''}, y_{r''}) = (x^*_r, y^*_r)$ traversing the path $p^*_{R''} = path(r_c, x^*_r) \cup (x^*_r, y^*_r) \cup path(y^*_r, r')$ (see Figure 4.3 (c)). Since y^*_r is higher than x', again the path from lowest edge on $p^*_{R''}$ from the component containing p_c, to p_1 would correspond to $p_{\tau'}$. This traversal is then followed by the traversal of $p_{R''} = p_{\tau'} \cup path(x', par(v_l))$ as shown in Figure 4.3 (c). Since $p_{R''}$ is a *disconnecting* traversal of τ' from p_1, the unvisited part of p_1, say p'_1, is not connected to the unvisited part of τ'. Also, p'_1 would not be connected to τ_d and hence component containing p_c, because y^*_r was the lowest edge above x' on p_1 from τ_d. Since the

unvisited part of τ' is not connected to p_c, $\mathcal{A}_1$ is satisfied. As described earlier, the component having p_c will have the new root in τ_d, being the only part of the component connected to $p_{R'}$. Since τ_d is not a heavy subtree, $\mathcal{A}_3$ is also satisfied.

Thus, in all the cases of Special Case of heavy path traversal, one of the traversals described above is necessarily applicable. Refer to Procedure Heavy-Special in Section 4.8 for pseudo code.

4.6 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.

4.7 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 (refer to Section 4.8 for the pseudo code). Moreover, each of these sets is an *independent* set of parallel queries on $\mathcal{D}$ (recall the definition of *independent* queries in Section 4.3). 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. In the parallel setting, these operations can be efficiently reduced to performing $O(1)$ sets of LCA queries on the DFS tree T using $|c|$ vertices (similar to *reduction* algorithm in Section 4.3) Refer to Section 4.9 for the details. Since our algorithm requires $\log n$ phases each having $\log n$ stages, we get the following theorem.

Theorem 4.2. *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 the semi-streaming or distributed model, it additionally performs some local computation requiring only the subtree τ. In the parallel model, it additionally performs $O(\log^2 n)$ sequential sets of $O(|\tau|)$ LCA queries on T.*

Procedure Reroot-DFS($r_c, p_c, \mathcal{T}_c$): Traversal enters through r_c into the component c containing a path p_c and set of trees $\mathcal{T}_c$.

```
/* Let current phase be P_i and current stage be S_j                    */
τ_c ← Heaviest tree in T_c;
T(v_H) ← Smallest subtree having size at least n/2^i;
if |τ_c| ≤ n/2^i then Return Reroot-DFS(r_c, p_c, T_c) in next phase;
if |p_c| ≤ n/2^j then Return Reroot-DFS(r_c, p_c, T_c) in next stage;
/* Disintegrating Traversal                                             */
if p_c = φ or r_c = root(τ_c) then Return DisInt-DFS(r_c, p_c, T_c);
/* Disconnecting Traversal                                              */
if r_c ∉ T_c ∪ {p_c} or r_c ∈ T(v_H) then Return DisCon-DFS(r_c, p_c, T_c);
if r_c ∈ p_c then Return Path-Halving(r_c, p_c, T_c, φ);   /* Path Halving */
Heavy-DFS(r_c, p_c, T_c) ;                    /* Heavy Subtree Traversal */
```

4.8 Pseudo codes of Traversals in Rerooting Algorithm

4.9 Implementation in the Parallel Environment

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.

4.9.1 Basic Data Structures

The data structure maintained by our algorithm uses the following classical results for finding the properties of a tree on an EREW PRAM.

Theorem 4.3 (Tarjan and Vishkin [TV84]). *A rooted tree on n vertices can be processed in $O(\log n)$ time using n processors to compute post order numbering of the tree, level and number of descendants for each vertex on a EREW PRAM.*

Theorem 4.4 (Schieber and Vishkin [SV88]). *A rooted tree on n vertices can be preprocessed in $O(\log n)$ time using n processors on an EREW PRAM such that k LCA queries can be answered in $O(1)$ time using k processors on a CREW PRAM.*

Using the standard simulation model [JáJ92] for converting a CRCW PRAM algorithm to EREW PRAM algorithm at the expense of extra $O(\log n)$ factor in the time complexity, we get the following theorem.

Theorem 4.5. *A rooted tree on n vertices can be preprocessed in $O(\log n)$ time using n processors on an EREW PRAM such that any k LCA queries can be answered in $O(\log n)$ time using k processors on an EREW PRAM.*

We also use the following classical result to sort and hence report maximum/minimum of a set of n numbers on an EREW PRAM.

Theorem 4.6 (Cole [Col88, Col93]). *A set of n numbers can be sorted using parallel merge sort in $O(\log n)$ time using n processors on an EREW PRAM.*

Procedure Process-Components($\mathcal{P}, \mathcal{T}, p^*$): Moves components created of type $C1$ and components created with $p \in \mathcal{P}$ of type $C2$ to the next stage, after traversal of $p^* = p_1 \cup p_2 \cup p_3$, the newly added path in T^*. Here, $p \in \mathcal{P}, p_1, p_2$ and p_3 are ancestor-descendant paths of T and traversal of p^* ensures components of type $C1$ and $C2$ with paths in $\mathcal{P}$ only.

foreach $p \in \mathcal{P}$ **do** /* $p = path(x, y)$, where x lower in T^* */
  foreach $\tau \in \mathcal{T}$ *in parallel using* $|\tau|$ *processors* **do**
    if $Query\big(\tau, path(x, y)\big) \neq \phi$ **then** /* $\exists$ edge from τ to p */
      $\mathcal{T} \leftarrow \mathcal{T} \setminus \{\tau\}$, $\mathcal{T}_p \leftarrow \mathcal{T}_p \cup \{\tau\}$;
    end
  end
  for $p' \in \{p_3, p_2, p_1\}$ **do** /* $p' = path(x', y')$, where x' lower in T^* */
    $\{x_p, y_p\} \leftarrow Query\big(p, path(x', y')\big)$; /* where $x_p \in p$ */
    foreach $\tau \in \mathcal{T}_p$ **do** $(x_\tau, y_\tau) \leftarrow Query\big(\tau, path(x', y')\big)$; /* where $x_\tau \in \tau$ */
    $\{x_p, y_p\} \leftarrow$ Lowest edge on T^* among (x_p, y_p) and $(x_\tau, y_\tau), \forall \tau \in \mathcal{T}_p$;
    if (x_p, y_p) *is a valid edge* **then** break;
  end
  Add (x_p, y_p) to T^*;
  Reroot-DFS$(x_p, p, \mathcal{T}_p)$ in current stage;
end
foreach $\tau \in \mathcal{T}$ *in parallel using* $|\tau|$ *processors* **do**
  $\mathcal{T} \leftarrow \mathcal{T} \setminus \tau$;
  for $p' \in \{p_3, p_2, p_1\}$ **do** /* $p' = path(x', y')$, where x' lower in T^* */
    $(x_\tau, y_\tau) \leftarrow Query\big(\tau, path(x', y')\big)$; /* where $x_\tau \in \tau$ */
    if (x_τ, y_τ) *is a valid edge* **then** break;
  end
  Add (y_τ, x_τ) to T^*;
  Reroot-DFS$(x_\tau, \phi, \{\tau\})$ in next stage;
end

Procedure DisInt-DFS($r_c, p_c, \mathcal{T}_c$): Disintegrating Traversal of a component c having a path p_c and a set of trees $\mathcal{T}_c$ through the root $r_c \in \tau \in \mathbb{T}_c$, where either $|p_c| = 0$ or $r_c = root(\tau)$.

$T(v_H) \leftarrow$ Smallest subtree τ' of τ, where $|\tau'| > n/2^i$;
$\mathcal{T} \leftarrow$ Subtrees hanging from $path\big(r_c, root(\tau)\big)$;
$T(v_h) \leftarrow$ Subtree from $\mathcal{T}$ containing v_H;
$\mathcal{T} \leftarrow \mathcal{T} \setminus T(v_h) \cup$ Subtrees hanging from $path(v_h, v_H)$;
Add $path(r_c, v_H)$ to T^*;
if $|p_c| \neq 0$ **then** $p \leftarrow p_c$; $\mathcal{T} \leftarrow \mathcal{T} \cup \mathcal{T}_c \setminus \tau$; /* Component of type $C2$ */
else $p \leftarrow path\Big(par\big(par(v_h)\big), root(\tau)\Big)$; /* remaining part of $path\big(r_c, root(\tau)\big)$ */
Process-Components$\big(\{p\}, \mathcal{T}, path(r_c, v_H)\big)$; /* Goes to DisCon-DFS or next phase */

Procedure Path-Halving-DFS$(r_c, p_c, \mathcal{T}_c)$: Traversal of a component c having a path p_c and a set of trees $\mathcal{T}_c$ through the root $r_c \in p_c$.

$p_c \leftarrow p_c \setminus path(r_c, x)$; `/*` $p_c = path(x, y)$ `where` $|path(x, r_c)| \geq |path(y, r_c)|$ `*/`
$p^* \leftarrow path(r_c, x)$;
Add p^* to T^*;
Process-Components$\big(\{p_c\}, \mathcal{T}_c, p^*\big)$; `/* Goes to next stage */`

Procedure DisCon-DFS$(r_c, p_c, \mathcal{T}_c)$: Disconnecting Traversal of a component c having a path p_c and a set of trees $\mathcal{T}_c$ through the root r_c, where either $r_c \in \tau \notin \mathbb{T}_c$ or $r_c \in T(v_H)$.

`/*` $p_c = path(u, v)$`, where` u `is ancestor of` v `*/`
if *τ has an edge to upper half of p_c* **then**
  $(x, y) \leftarrow$ Lowest edge from τ to p_c; $p'_c \leftarrow path(y, u)$; `/* where` $x \in \tau$ `*/`
else $(x, y) \leftarrow$ Highest edge from τ to p_c; $p'_c \leftarrow path(y, v)$; `/* where` $x \in \tau$ `*/`
$\mathcal{T} \leftarrow$ Subtrees hanging from $path\big(r_c, root(\tau)\big)$;
$T(v) \leftarrow$ Subtree from $\mathcal{T}$ containing x;
$\mathcal{T} \leftarrow \mathcal{T} \setminus T(v) \cup$ Subtrees hanging from $path(v, x)$;
$p \leftarrow path\Big(par\big(par(v)\big), root(\tau)\Big)$; `/* remaining part of` $path\big(r_c, root(\tau)\big)$ `*/`
$p^* \leftarrow path(r_c, x) \cup (x, y) \cup p'_c$;
Add p^*) to T^*;
Process-Components$\big(\{p, p_c \setminus p'_c\}, \mathcal{T} \cup \mathcal{T}_c \setminus \{\tau\}, p^*\big)$; `/* Goes to next stage/phase */`

Procedure Heavy-DFS($r_c, p_c, \mathcal{T}_c$): Heavy Subtree Traversal of a component c having a path p_c and a set of trees $\mathcal{T}_c$ through the root $r_c \in \tau \in \mathbb{T}_c$, where $r' = root(\tau)$

$T(v_H) \leftarrow$ Smallest subtree τ' of τ, where $|\tau'| > n/2^i$;
/* Considering Scenario 1. */
$\mathcal{T} \leftarrow$ Subtrees hanging from $path(r_c, r')$ with edge in p_c;
$T(v_L) \leftarrow$ Subtree from $\mathcal{T}$ containing v_H;
$p^* \leftarrow path(r_c, r')$;
$(x_1, y_1) \leftarrow$ Highest edge to p^* from $\tau' \in \mathcal{T}$ and p_c ; /* where $y_1 \in p^*$ */
if $x_1 \notin T(v_L)$ ***or*** $x_1 \in T(v_H)$ ***or*** $x_1 = v_L$ ***or*** $x_1 \in p_c$ **then**
  Add p^* to T^*;
  $\mathcal{T} \leftarrow$ Subtrees hanging from p^*;
  Return Process-Components$(\{p_c\}, \mathcal{T} \cup \mathcal{T}_c \backslash \tau, p^*)$;
  /* Goes to DisConn, DisInt or Path-Halving */
end
/* Considering Scenario 2. */
$\mathcal{T} \leftarrow \mathcal{T} \setminus T(v_L) \cup$ Subtrees hanging from $path(v_L, v_H)$ with edge in p_c;
$(x_d, y_d) \leftarrow$ Highest edge to p^* from $\tau' \in \mathcal{T}$; /* where $x_d \in \mathcal{T}'$ */
if $(x_d, y_d) = \phi$ **then** $y_d = r_c$;
$(x_p, y_p) \leftarrow \{(x', y') : x' \in T(v_L), y' \in path(y_d, r')$ of minimum $LCA(x', v_H)\}$;
$p^* \leftarrow path(r_c, x_p) \cup (x_p, y_p) \cup path(y_p, par(v_l))$;
$\mathcal{T} \leftarrow$ Subtrees hanging from $path(r_c, r')$ with edge in p_c;
$\mathcal{T} \leftarrow \mathcal{T} \setminus T(v_L) \cup$ Subtrees hanging from $path(v_L, x_p)$ with edge in p_c;
$(x_2, y_2) \leftarrow$ Lowest edge to p^* from $\tau' \in \mathcal{T}$ or p_c ; /* where $y_2 \in p^*$ */
$T(v_P) \leftarrow$ The subtree hanging from $path(v_L, x_p)$ having v_H;
if $x_2 \notin T(v_P)$ ***or*** $x_2 \in T(v_H)$ ***or*** $x_2 = v_P$ ***or*** $x_2 \in p_c$ **then**
  Add p^* to T^*;
  $\mathcal{T} \leftarrow$ Subtrees hanging from $path(r_c, r')$ and $path(v_L, x_p)$;
  Return Process-Components$(\{p_c, path(par(y_p), r')\}, \mathcal{T} \cup \mathcal{T}_c \backslash \tau, p^*)$;
  /* Goes to DisConn, DisInt or Path-Halving */
end
/* Considering Scenario 3. */
$\tau_d \leftarrow$ Subtree hanging on $path(v_L, v_H)$ having x_d;
$(x_2', y_2') \leftarrow$ Lowest edge from τ_d to (r_c, y_p);
if y_2 *lower than* y_2' **then** $(x_r, y_r) \leftarrow (x_2', y_2')$;
else $(x_r, y_r) \leftarrow (x_2, y_2)$;
$p^* \leftarrow path(r_c, x_r) \cup (x_r, y_r) \cup path(y_r, r')$;
$\mathcal{T} \leftarrow$ Subtrees hanging from $path(r_c, r')$ with edge to p_c;
$\mathcal{T} \leftarrow \mathcal{T} \setminus T(v_L) \cup$ Subtrees hanging from $path(v_L, x_r)$ with edge to p_c;
$(x_3, y_3) \leftarrow$ Lowest edge to p^* from $\tau' \in \mathcal{T}$ or p_c ; /* where $y_3 \in p^*$ */
$T(v_R) \leftarrow$ The subtree hanging from $path(v_L, x_r)$ having v_H;
if $x_3 \notin T(v_R)$ ***or*** $x_3 \in T(v_H)$ ***or*** $x_3 = v_P$ ***or*** $x_3 \in p_c$ **then**
  Add p^* to T^*;
  $\mathcal{T} \leftarrow$ Subtrees hanging from $path(r_c, r')$ and $path(v_L, x_r)$;
  Return Process-Components$(\{p_c, path(par(v_l), y_r) \setminus \{y_r\}\}, \mathcal{T} \cup \mathcal{T}_c \backslash \tau, p^*)$;
  /* Goes to DisConn, DisInt or Path-Halving */
end
Heavy-Special($r_c, p_c, \mathcal{T}_c$);

Procedure Heavy-Special($r_c, p_c, \mathcal{T}_c$): Special Case of Heavy Subtree Traversal of a component c having a path p_c and a set of trees $\mathcal{T}_c$ through the root $r_c \in \tau \in \mathbb{T}_c$.

/* Modified r' traversal. */
$p^*_{R'} \leftarrow path(r_c, x_2) \cup (x_2, y_2) \cup path\big(y_2, root(\tau)\big)$;
$p_1 \leftarrow path(par(v_l), y_2) \backslash \{y_2\}$;
$\mathcal{T} \leftarrow$ Subtrees hanging from $path\big(r_c, root(\tau)\big)$ with edge to p_1;
$\mathcal{T} \leftarrow \mathcal{T} \setminus T(v_L) \cup$ Subtrees hanging from $path(v_L, x_2)$ with edge to p_1;
$(x', y') \leftarrow$ Lowest edge to $p^*_{R'}$ from $\tau' \in \mathcal{T}$ or p_1 ; /* where $y' \in p^*_{R'}$ */
if y' *at most as high as* y_p **then**
  /* Root Traversal of τ_d. */
  $p_{R'} \leftarrow (y_p, x_p) \cup path(x_p, root(\tau_d))$;
  Add $p^*_{R'}$ and $p_{R'}$ to T^*;
  $\mathcal{T} \leftarrow$ Subtrees hanging from $path(r_c, root(\tau)) \backslash \{T(v_L)\}$;
  $\mathcal{T} \leftarrow \mathcal{T} \cup$ Subtrees hanging from $path(v_L, x_2) \backslash \{\tau_d\}$;
  $\mathcal{T} \leftarrow \mathcal{T} \cup$ Subtrees hanging from $path(x_d, root(\tau_d))$;
  Return Process-Components$\big(\{p_c, p_1\}, \mathcal{T} \cup \mathcal{T}_c \backslash \tau, p^*_{R'} \cup p_{R'}\big)$ /* Goes to DisConn, DisInt or Path-Halving */
end
/* Cover traversal of p_1 */
if $x' \notin p_1$ **then** /* Connected to p_1 through τ' */
  $y_{\tau'} \leftarrow x'$;
  $\{x', x_{\tau'}\} \leftarrow$ Highest edge on p_1 from τ' ; /* where $x' \in p_1$ */
end
if x' *at most as high as* y_r *on* p_1 **then**
  /* Upward Cover Traversal of p_1. */
  if $x_{\tau'} \neq \phi$ **then** $\{x', x_{\tau'}\} \leftarrow$ Lowest edge on p_1 from τ' /* where $x' \in p_1$ */ ;
  $p_{R'} \leftarrow p_{\tau'} \cup path(x', y_2) \backslash \{y_2\}$;
  $p'_1 \leftarrow path(par(v_l), x') \backslash \{x'\}$;
  $\mathcal{T} \leftarrow$ Subtrees hanging from $path(v_L, x_2)$;
else
  /* Lower Cover Traversal of p_1. */
  $(x^*_r, y^*_r) \leftarrow$ Highest edge from τ_d to $path(par(y_r), y_2) \backslash \{y_2\}$; /* where $y^*_r \in path(y_r, y_2)$ */
  if $(x^*_r, y^*_r) = \phi$ **then** $(x^*_r, y^*_r) \leftarrow (x_2, y_2)$;
  $p^*_{R'} \leftarrow path(r_c, x^*_r) \cup (x^*_r, y^*_r) \cup path\big(y^*_r, root(\tau)\big)$;
  $p_{R'} \leftarrow p_{\tau'} \cup path(x', par(v_l))$;
  $p'_1 \leftarrow path(par(x'), y^*_r) \backslash \{y^*_r\}$;
  $\mathcal{T} \leftarrow \mathcal{T} \cup$ Subtrees hanging from $path(v_L, x^*_r)$;
end
if $x_{\tau'} \neq \phi$ **then** /* Connected to p_1 through τ' */
  $p_{\tau'} \leftarrow (y', y_{\tau'}) \cup path(y_{\tau'}, x_{\tau'}) \cup (x_{\tau'}, x')$;
  $p^*_{\tau'} \leftarrow path(LCA(x_{\tau'}, y_{\tau'}), root(\tau'))$;
  $\mathcal{T}_{\tau'} \leftarrow$ Subtrees hanging from $path(x_{\tau'}, y_{\tau'})$ and $path(LCA(x_{\tau'}, y_{\tau'}), root(\tau'))$;
else $p_{\tau'} \leftarrow (y', x')$; $p^*_{\tau'} = \phi$; $\mathcal{T}_{\tau'} = \phi$;
Add $p^*_{R'}$ and $p_{R'}$ to T^*;
$\mathcal{T} \leftarrow \mathcal{T} \cup \mathcal{T}_{\tau'} \cup$ Subtrees hanging from $path(r_c, root(\tau)) \backslash \{T(v_L)\}$;
Return Process-Components$\big(\{p_c, p'_1, p^*_{\tau'}\}, \mathcal{T} \cup \mathcal{T}_c \backslash \tau, p^*_{R'} \cup p_{R'}\big)$;
/* Goes to DisConn, DisInt or Path-Halving */

4.9.2 Implementation of operations on T

As described earlier several properties of T can be reported in $O(1)$ time using the data structures described in Theorem 4.3 and Theorem 4.5.

1. **Determine whether an edge (x, y) is a back edge in T**
   This query can easily be answered by finding $l = LCA(x, y)$. If $l = x$ or $l = y$ the edge (x, y) is a back edge in T. Hence, reporting whether an edge is a back edge can be reduced to finding LCA of two vertices in T.

2. **Finding length of a path**
   Compare the level of the two endpoints as reported by structure in Theorem 4.3.

3. **Given $x \in T(y)$, find child y' of y such that $x \in T(y')$**
   For each vertex v of the graph perform the following in parallel (using $|T(y)|$ processors): if $par(v)$ is y and $LCA(v, x)$ is v then report v. This query too reduces to finding LCA of two vertices in T.

4. **Determine whether x lies on $path(y, z)$, where y is ancestor of z**
   If $LCA(x, z) = x$ and $LCA(x, y) = y$, then x lies on $path(y, z)$.

5. **Finding subtrees hanging from a $path(x, y)$, where x is ancestor of y**
   For each vertex v of the graph perform the following in parallel (using total n processors), if $LCA(v, y) = par(v)$ then $T(v)$ is a subtree hanging from the path.

The number of processors required for the last three queries is equal to the size of the corresponding component, remaining queries requiring a single processor each. Thus, using Theorem 4.6 and procedures described above we have the following theorem

Theorem 4.7. *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 incident on a path, using k processors per query, where k is the size of the corresponding component.*

4.9.3 Implementation of $\mathcal{D}$

Given the DFS tree T of the graph, we build the data structures described in Theorem 4.3 and Theorem 4.5 on it. Now, given the post order traversal of T, we assign each vertex with a value equal to its rank in the post order traversal. Now, for each vertex v we perform a parallel merge sort on the set of neighbors of the vertex using $degree(v)$ processors, requiring overall m processors. Thus, each vertex stores its neighbors $\big($say $N(v)\big)$ in the increasing order of their post order indexes. Due to absence of cross edges in a DFS tree T, the neighbors of every vertex would be sorted in the order they appear on the path from $root(T)$ to the vertex. Thus, the data structure $\mathcal{D}$ can be built in $O(\log n)$ time (for

sorting) using m processors on an EREW PRAM. This allows us to answer the following queries efficiently.

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 $path(x,y)$.

We now describe how to perform a set of *independent* queries to $\mathcal{D}$ (recall definition of *independent* queries in Section 4.3) in $O(\log n)$ time on an EREW PRAM as follows. We assign one processor for each vertex $u \in \{w\}, T(w)$ or $path(x,y)$ (depending on the type of query) to perform the following in parallel. For the vertex u, we would first perform a binary search for the range given by the post order indexes of x and y on $N(u)$ to find the required edge. However, since all vertices of $path(x,y)$ may not be ancestors of u, $N(u)$ may include some edges not on $path(x,y)$ too in the given range, corrupting the search results. Hence, the search would be performed on a modified range described as follows. Firstly, assuming x is an ancestor of y, if $LCA(u,x)$ is not equal to x the search would not be performed (as x is not an ancestor of u). Otherwise, the search is performed on the range given by post order indexes of x and $LCA(u,y)$. However, in case of $Query\big(path(v,w), path(x,y)\big)$ we surely know that no vertex of $path(v,w)$ is a descendant of $path(x,y)$ (recall its definition in Section 4.3). Thus, we reverse the roles of the paths taking maximum or minimum accordingly using $|path(x,y)|$ processors. Thus, each of these queries would require $O(\log n)$ time on an EREW PRAM. Now, given a set of *independent* queries on $\mathcal{D}$, each processor shall be using different $N(u)$ for finding the corresponding edge. Hence, all the queries can be performed simultaneously on different memory cells abiding the constraints of an EREW PRAM. Now, the highest or lowest edge among all the edges returned by different processors can be found by taking the maximum or minimum in $O(\log n)$ time on an EREW PRAM (Theorem 4.6). Thus, we have the following theorem.

Theorem 4.8. *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 $Query\big(w, path(x,y)\big)$, $Query\big(T(w), path(x,y)\big)$ and $Query\big(path(v,w), path(x,y)\big)$ 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.*

Extension to handle 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 $\mathcal{D}_i$. We now show that any query of the type $Query\big(w, path(x,y)\big)$, $Query\big(T_i^*(w), path(x,y)\big)$ and $Query\big(path(v,w), path(x,y)\big)$ on $\mathcal{D}_i$, can be performed on $\mathcal{D}_0$ if $path(x,y)$ is an ancestor-descendant path in T. Recall that each such query is performed by querying the $N(x)$ corresponding to each descendant vertex x separately, whose results are later combined. Thus, even if $T_i^*(w)$ is not a subtree of T or $path(v,w)$

is not an ancestor-descendant path of T, it does not affect the processing of the query, as long as $path(x, y)$ is an ancestor-descendant path of T.

The only extra procedure to be performed to answer such queries correctly using $\mathcal{D}_0$, is to update the $N(x)$ for any vertex x whose adjacency list is affected by the graph update. For insertion/deletion of a vertex x, we simply add/delete the corresponding list $N(x)$. For insertion of vertex we additionally sort it according to post order traversal of T using n processors in $O(\log n)$ time. Note that we do not need to update the $N(y)$ for each neighbor y of x, as the query path being an ancestor-descendant path of both T_i^* and T would not contain x. However, on insertion of a vertex x, such a query can be made with the entire path representing only x. Hence, we assign the highest post order number to x, and add it to the end of $N(y)$ for each neighbor y of x. This can be done using n processors in $O(1)$ time on an EREW PRAM. Insertion/deletion of single edges can be taken care of individually by each search procedure taking $O(\log n + k)$ time to perform search after k updates. Thus, we have the following theorem.

Theorem 4.9. *The data structure $\mathcal{D}$ built on the DFS tree T of a graph G, can be used to perform a set of independent queries on $\mathcal{D}_k$ of types $Query\big(w, path(x, y)\big)$, $Query\big(T_k^*(w), path(x, y)\big)$ and $Query\big(path(v, w), path(x, y)\big)$, in $O(\log n + k)$ time using $1, |T_k^*(w)|$ and $|path(x, y)|$ processors respectively on an EREW PRAM, if $path(x, y)$ is an ancestor-descendant path of T.*

4.9.4 Analysis

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 4.1 reduces to the following theorem.

Theorem 4.10. *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 3.3 and Theorem 4.7, 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 (refer to pseudo code in Section 4.8) are trivially reducible to the operations described in Theorem 4.7. 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 4.11. *Given an undirected graph with the data structure $\mathcal{D}$ built 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 3.3, Theorem 4.10 and Theorem 3.1, we can prove our main result described as follows.

Theorem 4.12. *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 [Tar72] or any advanced deterministic parallel algorithm [AA88, GPV93]. 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 4.8 reducing Theorem 4.12 to the following theorem.

Theorem 4.13 (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 4.8). Thus, we build the data structure $\mathcal{D}$ using Theorem 4.8 during preprocessing itself, and attempt to use it to handle multiple updates.

Parallel fault tolerant DFS with 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 Section 4.9.3). 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 $O(\log^{2i-1} n)$ time on an EREW PRAM using n processors when $k \leq \log n$ (see Theorem 4.8 and Section 4.9.3). The other data structures on T_{i-1}^* can be built in $O(\log n)$ time using n processors (see Theorem 4.7). This allows our algorithm to build the DFS tree T_i^* from T_{i-1}^* using $\mathcal{D}$ in $O(\log^{2i+1} n)$ time on an EREW PRAM using n processors (see Theorem 4.2). 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 4.14 (Parallel Fault Tolerant DFS)**.** *Given an undirected graph, it can be pre-processed 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 in Section 3.3. However, this algorithm uses much simpler data structure $\mathcal{D}$ at the cost of a more complex algorithm.*

4.10 Applications in other models of computation

We now briefly describe how our algorithm can be easily adopted to the semi-streaming model and distributed model.

4.10.1 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 4.1 and Theorem 4.2) 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, the algorithm is first executed for all the components in turn until each instance of the algorithm queries the data structure $\mathcal{D}$. This is followed by a pass on the input graph to answer these queries and so on. Since each stage requires $O(1)$ steps (and hence $O(1)$ sequential queries on $\mathcal{D}$), it can be performed using $O(1)$ passes. Thus, our algorithm requires $O(\log^2 n)$ passes to update the DFS tree after a graph update by executing $\log n$ stages for each of the $\log n$ phases. Thus, we get the following theorem.

Theorem 4.15. *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.*

4.10.2 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 4.1 and Theorem 4.2, 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 [Pel00], we can propagate these $O(n)$ words in $O(D)$ rounds using messages of size $O(n/D)$, where D is the diameter of the graph. We now describe how our algorithm can be efficiently implemented in the distributed model.

Implementation details

In the synchronous $\mathcal{CONGEST}(B)$ model a processor is present at every node of the graph and communication links are restricted to the edges of the graph. The communication occurs in synchronous rounds, where each nodes can send a message of $O(B)$ words along each communication link. Our model includes a *preprocessing* stage followed by an alternating sequence of *update* and *recovery* stages. The graph is updated in the *update* stage, after which the *recovery* stage starts in which the algorithm updates the DFS tree of the graph. The model allows the algorithm to complete updating the DFS tree (completing the recovery stage) before the next update is applied to the graph (update stage). Similar model was earlier used by Henzinger et al. [HKN13]. We use an additional constraint of a space restriction of $O(n)$ size at each node. In the absence of 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. Finally, we also allow the deletion updates to be *abrupt*, i.e., the deleted link/node becomes unavailable for use instantly after the update.

Each node stores the current DFS tree T and the partially built DFS tree T^*. Thus, all the operations on T can be performed locally at each node, where the distributed computation is used only to evaluate the queries on $\mathcal{D}$. Also, using Theorem 4.1 and Theorem 4.2 each update reduces to $O(\log^2 n)$ sequential sets of $O(n)$ independent queries on $\mathcal{D}$. Thus, we shall only focus on how to evaluate such queries efficiently in the distributed environment.

Optimality of message size

We first prove that any distributed algorithm maintaining the DFS tree at each node requires a message size of $\Omega(n/D)$ to update the DFS tree in $O(D)$ rounds. Consider the insertion of a vertex, such that the final DFS tree uses $O(n)$ of the newly inserted edges. This is clearly possible if the current DFS tree has $O(n)$ branches, where leaf of each branch is connected to the inserted vertex. Thus, the information of at least these $O(n)$ new edges needs to be propagated throughout the network by any algorithm maintaining DFS tree at each node. Now, broadcasting m messages on a network with diameter D requires $\Omega(m + D)$ rounds [Pel00]. In order to limit the number of rounds to $O(D)$, we can send only $O(D)$ messages. Thus, any algorithm sending $O(n)$ words of information using $O(D)$ messages would require a message size of $\Omega(n/D)$. We thus use the $\mathcal{CONGEST}(n/D)$ model for our distributed algorithm.

Evaluation of queries on $\mathcal{D}$

Now, each node only stores the adjacency list of the corresponding vertex in addition to T and T^* described above. Recall that a query on $\mathcal{D}$ is merely highest/lowest edge among a set of eligible edges. Hence, it can be easily evaluated for the whole graph by combining the partial solutions of the same query performed on each adjacency list locally at the node. Thus, the focus is to broadcast the partial solution from each node to reach the whole graph, where each node can then combine them locally to get the solution to the query. Moreover, these partial solutions can also be combined during broadcasting itself to avoid sending too many messages as described below.

Performing broadcasts efficiently

Broadcasts can be performed efficiently by using a spanning tree of the graph. To ensure efficiency of rounds we use a BFS tree as follows. After every update, any vertex (say vertex with the smallest index) starts building a BFS tree $\mathcal{B}$ rooted at it. The depth of $\mathcal{B}$ is $O(D)$ and it can be built in $O(D)$ rounds using $O(m)$ messages [Pel00]. All the broadcasts are now performed only on the tree edges of $\mathcal{B}$ as follows. We first describe it for a single query then extend it to handle $O(n)$ queries. Note that it is a trivial extension of the standard pipelined broadcasts and convergecasts algorithm [Pel00]. Each node waits for partial solutions to the query from all its children in $\mathcal{B}$, updates its solution and sends it to its parent. On receiving the partial solutions from all the children, the root computes the final solution and sends it back to all nodes along the tree edges of $\mathcal{B}$. Clearly, this process requires $O(D)$ rounds and $O(n)$ messages each of size $O(1)$ (partial solution of a query is a single edge). In order to perform $O(n)$ independent queries efficiently in parallel, on each edge we send D messages of size $O(n/D)$ in a pipelined manner (one after the other) to achieve the broadcast in $O(D)$ time (see pipelined broadcast in [Pel00]). The total number of messages sent would be $O(nD)$. Since the rerooting algorithm requires $O(\log^2 n)$ sequential sets of $O(n)$ queries (see Theorem 4.2), we get the following theorem.

Theorem 4.16. *Given any arbitrary online sequence of vertex or edge updates, we can maintain a DFS tree in $O(D \log^2 n)$ rounds per update in a distributed setting using $O(nD \log^2 n + m)$ messages each of size $O(n/D)$ and $O(n)$ local space on each processor, where D is diameter of the graph.*

Remark. *Our initial assumption of adding a pseudo root (see Appendix A) connected to every vertex of the graph is no longer valid in the distributed system. This is because both the processors and communication links are fixed in our model. Thus, we need to maintain a DFS forest instead of a DFS tree requiring to handle the cases when some component is partitioned into several components and when two or more components merge as a result of a graph update. The following section describes how this can be achieved in the same bounds described above.*

Maintaining a DFS forest

After every update in the graph, a neighboring vertex of the affected link/node shall broadcast the information about the update to all the vertices in the component. However, in order to limit the number of messages transmitted, exactly *one* vertex from each component so formed needs to initiate the broadcast. We shall shortly describe how to choose this vertex. The chosen vertex also chooses the new root for the DFS tree of the component (say the node with the smallest index). The new root then makes the corresponding BFS tree as described above to perform efficient broadcasts. In case two or more components are merged due to the update, the DFS tree of each component computed earlier is broadcasted to the entire component by the original roots of two components. Since, the size of broadcast (DFS tree) is $O(n)$, it can be performed under the same bounds as described above.

We now describe how to choose the broadcast vertex efficiently. In case of vertex/edge insertion, we choose the inserted vertex or endpoint of the inserted edge with smaller index respectively. In case of vertex/edge deletion, for each component so formed, we choose

the neighbor of deleted node/link in T that has the smallest index. For this each neighbor of the deleted node/link needs to know the resultant components formed as a result of the deletion. This can be easily computed locally if each node also stores the articulation points/bridges of the current DFS tree T. Hence, after computing the DFS tree, each node computes the articulation points/bridges of the DFS tree according of the subgraph induced by the edges of T and the adjacency list stored at the node. The vertices/edges present in *all* the sets of articulation points/bridges computed at different nodes will be the articulation points/bridges of the whole graph. Again, this requires each vertex to send $O(n)$ words of information where the partial solutions can be combined. Thus, it can be performed similar to the queries on $\mathcal{D}$ using the same bounds.

4.11 Discussion

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 (see Chapter 3) is $\tilde{O}(\sqrt{mn})$. Even though our fault tolerant algorithm is nearly work optimal, it is 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 in Chapter 3 for 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.

Chapter 5

Empirical analysis of Incremental DFS algorithms

5.1 Introduction

In the past chapters, we have seen several algorithms that maintain a DFS tree in the dynamic setting. However, not much is known about their empirical performance and hence their applicability in practice. For various algorithms, the average-case time complexity (average performance on random graphs) have been proven to be much less than their worst case complexity. Another equally important aspect is the empirical performance of an algorithm on real world graphs. After all, the ideal goal is to design an algorithm having a theoretical guarantee of efficiency in the worst case as well as superior performance on real graphs. Often such an experimental analysis also leads to the design of simpler algorithms that are extremely efficient in real world applications. Thus, such an analysis bridges the gap between theory and practice.

In this chapter, we carry out extensive experimental and theoretical evaluation of the existing algorithms for maintaining incremental DFS in random graphs and real world graphs. Our study focuses on only incremental DFS algorithms for two reasons. *Firstly,* most dynamic graphs in real world are dominated by insertion updates [Kun16, LK14, DH14]. *Secondly*, in every other dynamic setting, only a single dynamic DFS algorithm is known, making a comparative study impractical.

For undirected graphs, there are two prominent algorithms, namely, ADFS1 and ADFS2, presented in Chapter 2. Chapter 3 also presents an incremental algorithm, namely WDFS, having better worst case guarantees on the update time. For directed acyclic graphs, the only non-trivial algorithm, namely FDFS, was presented by Franciosa et al. [FGN97]. However, even after 20 years of this result, there does not exist any non-trivial incremental DFS algorithm for directed graphs with $o(m^2)$ worst case bound. In the following section we briefly describe various algorithms considered in our evaluation.

5.2 Existing algorithms

In this section we give a brief overview of the results on maintaining incremental DFS (see Table 5.1 for comparison). The key ideas used in these algorithms are crucial to understand their behavior on random graphs.

Static DFS algorithm (SDFS)

The static algorithm for computing the DFS tree of a graph was given by Tarjan [Tar72]. In the incremental version of the same, SDFS essentially computes the whole DFS tree from scratch after every edge insertion.

Static DFS algorithm with interrupt (SDFS-Int)

Static DFS tree was shown to have much better performance for a random graph by Kapidakis [Kap90]. Only difference from SDFS is that the algorithm terminates as soon as all the vertices of the graph are marked visited. Again, the algorithm recomputes the DFS tree from scratch after every edge insertion though requiring only $O(n \log n)$ time for random graphs.

Incremental DFS for DAG/directed graph (FDFS)

FDFS [FGN97] maintains the post-order (or DFN) numbering] of vertices in the DFS tree, which is used to rebuild the DFS tree efficiently. On insertion of an edge (x, y) in the graph, it first checks whether (x, y) is an anti-cross edge by verifying if DFN[x] <DFN[y]. In case (x, y) is not an anti-cross edge, it simply updates the graph and terminates. Otherwise, it performs a partial DFS on the vertices reachable from y in the subgraph induced by the vertices with DFN number between DFN[x] and DFN[y]. In case of DAGs, this condition essentially represents a *candidate set* of vertices that lie in the subtrees hanging on the right of $path(LCA(x, y), x)$ or on the left of $path(LCA(x, y), y)$. FDFS thus removes these reachable vertices from the corresponding subtrees and computes their DFS tree rooted at y to be hanged from the edge (x, y). The DFN number of all the vertices in candidate set is then updated to perform the next insertion efficiently. The algorithm can also be trivially extended to directed graphs. Here, the *candidate* set includes the subtrees hanging on the right of $path(LCA(x, y), x)$ until the entire subtree containing y (say T'). Note that for DAGs instead of entire T', just the subtrees of T' hanging on the left of $path(LCA(x, y), y)$ are considered. However, FDFS in directed graphs is not known to have any bounds better than $O(m^2)$.

Incremental DFS for undirected graphs (ADFS)

In Chapter 2, we presented two algorithms (referred as ADFS1 and ADFS2) for maintaining incremental DFS in undirected graphs. ADFS (refers to both ADFS1 and ADFS2) maintains a data structure that answers LCA and level ancestor queries. Recall that on insertion of an edge (x, y) in the graph, ADFS first verifies whether (x, y) is a cross edge. In case (x, y) is a back edge, it simply updates the graph and terminates. Otherwise, it rebuilds the DFS tree using a simple procedure that may convert several back edges of the tree to cross edges. It then collects these cross edges and iteratively inserts them back to the graph using the same procedure. The only difference between ADFS1 and ADFS2 is the order in which these collected cross edges are processed. ADFS1 processes these edges arbitrarily, whereas ADFS2 processes the cross edge with the highest endpoint first. For this purpose ADFS2 uses a non-trivial data structure, which shall be referred as $\mathcal{D}$.

Incremental DFS with worst case guarantee (WDFS)

In Chapter 3, we presented an incremental DFS algorithm giving a worst case guarantee of $O(n \log^3 n)$ on the update time. The algorithm builds a data structure using the current DFS tree, which is used to efficiently rebuild the DFS tree after an edge update. Recall that building this data structure requires $O(m)$ time and hence the same data structure is used to handle multiple updates ($\approx \tilde{O}(m/n)$). The data structure is then rebuilt over a period of updates using a technique called *overlapped periodic rebuilding*. Now, the edges processed for updating a DFS tree depends on the number of edges inserted since the data structure was last updated. Thus, whenever the data structure is updated, there is a sharp fall in the number of edges processed per update resulting in a saw like structure on the plot of number of edges processed (or time taken) per update.

Algorithm	**Graph**	**Time per update**	**Total time**
SDFS [Tar72]	Any	$O(m)$	$O(m^2)$
SDFS-Int [Kap90]	Random	$O(n \log n)$ expected	$O(mn \log n)$ expected
FDFS [FGN97]	DAG	$O(n)$ amortized	$O(mn)$
ADFS1 (Ch. 2)	Undirected	$O(n^{3/2}/\sqrt{m})$ amortized	$O(n^{3/2}\sqrt{m})$
ADFS2 (Ch. 2)	Undirected	$O(n^2/m)$ amortized	$O(n^2)$
WDFS (Ch. 3)	Undirected	$O(n \log^3 n)$	$O(mn \log^3 n)$

Table 5.1: Comparison of different algorithms for maintaining incremental DFS of a graph.

5.3 Preliminary

5.3.1 Random Graphs

The two prominent models for studying random graphs are $G(n, m)$ [Bol84] and $G(n, p)$ [ER59, ER60]. A random graph $G(n, m)$ consists of the first m edges of a uniformly random permutation of all possible edges in a graph with n vertices. In a random graph $G(n, p)$, every edge is present in the graph with a probability of p independent of other edges. We now state the following classical result for random graphs that shall be used in our analysis.

Theorem 5.1. *[FK15] Graph $G(n, p)$ with $p = \frac{1}{n}(\log n + c)$ is connected with probability at least $1 - e^{-c}$ for any constant $c > 0$.*

5.3.2 Experimental Setting

In our experimental study on random graphs, the performance of different algorithms is analyzed in terms of the number of edges processed, instead of the time taken. This is because the total time taken by the evaluated algorithms is dominated by the time taken to process the graph edges. The following short discussion shall throw a better light at our choice of edges processed for analyzing the algorithms.

Most of the algorithms analyzed in this chapter require dynamic maintenance of a data structure for answering LCA and LA (level ancestor) queries. The LCA/LA data structures used by ADFS1/ADFS2 (Ch. 2) takes $O(1)$ amortized time to maintain the data structure for every vertex whose ancestor is changed in the DFS tree. However, it

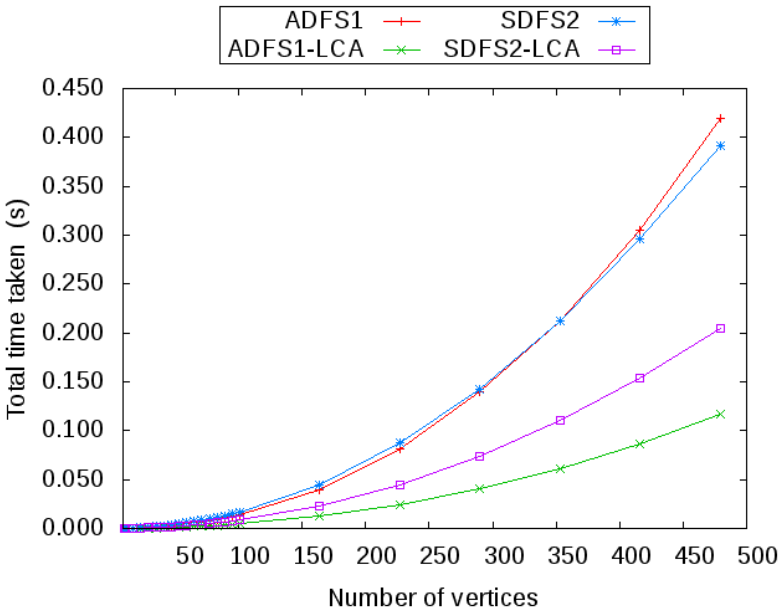

Figure 5.1: Comparison of total time taken and time taken by LCA/LA data structure by the most efficient algorithms for insertion of $m = \binom{n}{2}$ edges for different values of n.

is quite difficult to implement and seems to be more of theoretical interest. Thus, we use a far simpler data structure whose maintenance require $O(\log n)$ time for every vertex whose ancestor is changed in the DFS tree. Figure 5.1 shows that the time taken by these data structures is insignificant in comparison to the total time taken by the algorithm. Analyzing the number of edges processed instead of time taken allows us to ignore the time taken for maintaining and querying this LCA/LA data structure. Moreover, the performance of ADFS and FDFS is directly proportional to the number of edges processed along with some vertex updates (updating DFN numbers for FDFS and LCA/LA structure for ADFS). However, the tasks related to vertex updates can be performed in $\tilde{O}(1)$ time using dynamic trees [Tar97]. Thus, the actual performance of these algorithms is truly depicted by the number of edges processed, justifying our evaluation of relative performance of different algorithms by comparing the number of edges processed.

Further, comparing the number of edges processed provides a deeper insight in the performance of the algorithm (see Section 5.4). Also, it makes this study independent of the computing platform making it easier to reproduce and verify. For random graphs, each experiment is averaged over several test cases to get the expected behavior. For the sake of completeness, the corresponding experiments are also replicated measuring the time taken by different algorithms in Section 5.9. However, for real graphs the performance is evaluated by comparing the time taken and not the edges processed. This is to ensure an exact evaluation of the relative performance of different algorithms. The source code of our project is available on Github under the BSD 2-clause license[1].

[1]https://github.com/shahbazk/IncDFS-Experimental

5.3.3 Datasets

In our experiments we considered the following types of datasets.

- **Random Graphs:** The initial graph is the star graph, formed by adding an edge from the pseudo root s to each vertex. The update sequence is generated based on Erdős Rényi $G(n, m)$ model by choosing the first m edges of a random permutation of all the edges in the graph. For the case of DAGs, the update sequence is generated using an extension of $G(n, m)$ model for DAGs [CMP+10].
- **Real graphs:** We use a number of publicly available datasets [Kun16, LK14, DH14] derived from the real world. These include graphs related to Internet topology, collaboration networks, online communication, friendship networks and other interactions (for details refer to Section 5.7.3).

5.4 Experiments on Random Undirected graphs

We now compare the empirical performance of the existing algorithms for incrementally maintaining a DFS tree of a random undirected graph described in Section 5.2.

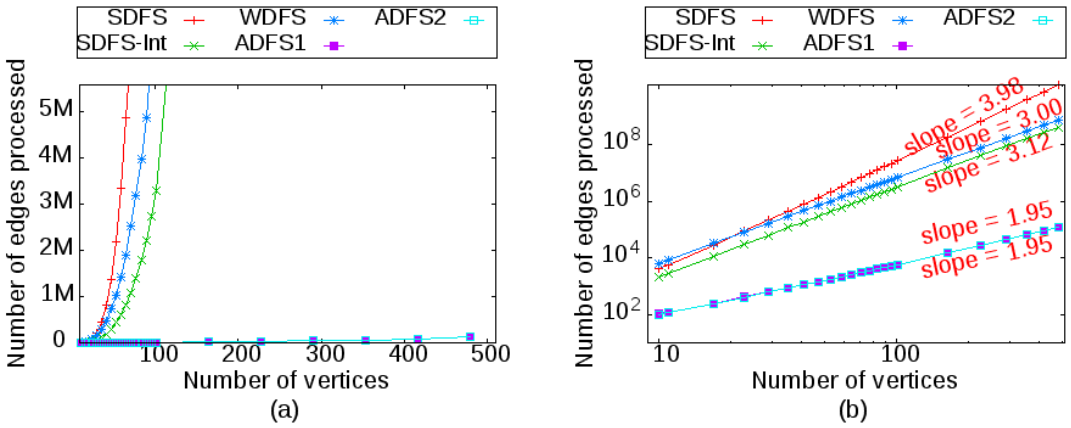

Figure 5.2: Total number of edges processed by existing algorithms for insertion of $m = \binom{n}{2}$ edges for different values of n. (a) Normal scale. (b) Logarithmic scale. See Figure 5.13 for corresponding time plot.

We first compare the total number of edges processed by the existing algorithms for insertion of $m = \binom{n}{2}$ edges, as a function of number of vertices. Figure 5.2 (a) and Figure 5.2 (b) shows this comparison in normal and logarithmic scale respectively. In Figure 5.2 (b), since the total number of edges is presented in logarithmic scale, the slope x of a line depicts the growth of the total number of edges as $O(n^x)$. The performance of SDFS, SDFS-Int and WDFS resemble their asymptotic bounds described in Table 5.1. For small values of n, WDFS performs worse than SDFS and SDFS-Int because of large difference between the constant terms in their asymptotic bounds, which is evident from their y-intercepts. However, the effect of constant term diminishes as the value of n is increased. The most surprising aspect of this experiment is the exceptional performance of ADFS1 and ADFS2. Both ADFS1 and ADFS2 perform much faster than the other algorithms. Furthermore, ADFS1 and ADFS2 perform equally well despite the difference in their asymptotic complexity (see Table 5.1).

Inference I_1: ADFS1 and ADFS2 perform equally well and much faster than other algorithms.

Remark: Inference I_1 is surprising because the complexity of ADFS1 and ADFS2 has been shown (see Chapter 2) to be $O(n^{3/2}\sqrt{m})$ and $O(n^2)$ respectively. Moreover, we present a sequence of m edge insertions where ADFS1 takes $\Omega(n^{3/2}\sqrt{m})$ time in Section 2.5.3, proving the tightness of its analysis. However, ADFS2 takes slightly more time than ADFS1, for maintaining the data structure $\mathcal{D}$ (see Figure 5.13).

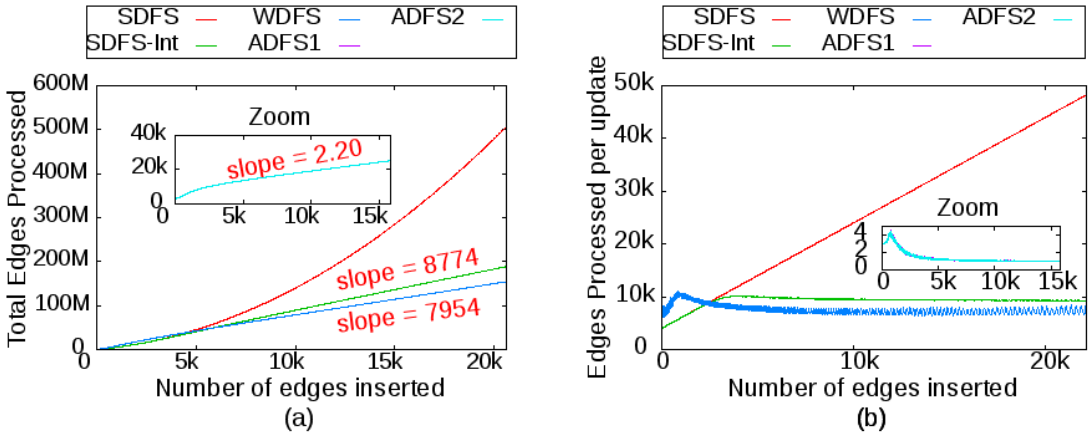

Figure 5.3: For $n = 1000$ and up to $n\sqrt{n}$ edge insertions the plot shows (a) Total number of edges processed, (b) Number of edges processed per edge insertion, by the existing algorithms. See Figure 5.14 for corresponding time plot.

We now compare the total number of edges processed by the existing algorithms as a function of number of inserted edges in Figure 5.3 (a). The slopes of SDFS-Int, WDFS and ADFS represent the number of edges processed per edge insertion. Here again, the performance of SDFS, SDFS-Int and WDFS resembles with their worst case values (see Table 5.1). Similarly, both ADFS1 and ADFS2 perform equally well as noted in the previous experiment. When the graph is sparse ($m << n\log^3 n$), WDFS performs worse than SDFS because of high cost of update per edge insertion (see Table 5.1). Further, as expected the plots of SDFS-Int and WDFS grow linearly in m. This is because their update time per insertion is independent of m. However, the plots of ADFS1 and ADFS2 are surprising once again, because they become almost linear as the graph becomes denser. In fact, once the graph is no longer sparse, each of them processes ≈ 2 edges per edge insertion to maintain the DFS tree. This improvement in the efficiency of ADFS1 and ADFS2 for increasing value of m is counter-intuitive since more edges may be processed to rebuild the DFS tree as the graph becomes denser.

Inference I_2: ADFS1/ADFS2 processes ≈ 2 edges per insertion after the insertion of $O(n)$ edges.

Finally, to investigate the exceptional behavior of ADFS1 and ADFS2, we compare the number of edges processed per edge insertion by the existing algorithms as a function of number of inserted edges in Figure 5.3 (b). Again, the expected behavior of SDFS, SDFS-Int and WDFS matches with their worst case bounds described in Table 5.1. The plot of WDFS shows the saw like structure owing to *overlapped periodic rebuilding* of the data structure used by the algorithm (see Section 5.2). Finally, the most surprising result of the experiment is the plot of ADFS1 and ADFS2 shown in the zoomed component of the plot. The number of edges processed per edge insertion sharply increases to roughly 5 (for $n = 1000$) when m reaches $O(n)$ followed by a sudden fall to reach 1 asymptotically. Note that the inserted edge is also counted among the processed edges, hence essentially the number of edges processed to update the DFS tree asymptotically reaches zero as the graph becomes dense. This particular behavior is responsible for the exceptional performance of ADFS1 and ADFS2.

Inference I_3: Number of edges processed by ADFS1/ADFS2 for updating the DFS tree asymptotically reaches zero as the graph becomes denser.

To understand the exceptional behavior of ADFS1 and ADFS2 for random graphs inferred in I_1, I_2 and I_3, we shall now investigate the structure of a DFS tree for random graphs.

5.5 Structure of a DFS tree: The broomstick

We know that SDFS, SDFS-Int and WDFS invariably rebuild the entire DFS tree on insertion of every edge. We thus state the first property of ADFS that differentiates it from other existing algorithms.

Property P_1: ADFS rebuilds the DFS tree only on insertion of a cross edge.

Let T be any DFS tree of the random graph $G(n, m)$. Let p_c denote the probability that the next randomly inserted edge is a cross edge in T. We first perform an experimental study to determine the behavior of p_c as the number of edges in the graph increases. Figure 5.4 (a) shows this variation of p_c for different values of n. The value p_c starts decreasing sharply once the graph has $\Theta(n)$ edges. Eventually, p_c asymptotically approaches 0 as the graph becomes denser. Surely ADFS crucially exploits this behavior of p_c in random graphs (using Property P_1). In order to understand the reason behind this behavior of p_c, we study the structure of a DFS tree of a random graph.

5.5.1 Broomstick Structure

The structure of a DFS tree can be described as that of a broomstick as follows. From the root of the DFS tree there exists a downward path on which there is no branching, i.e., every vertex has exactly one child. We refer to this path as the *stick* of the broomstick structure. The remaining part of the DFS tree (except the *stick*) is called the *bristles* of the broomstick.

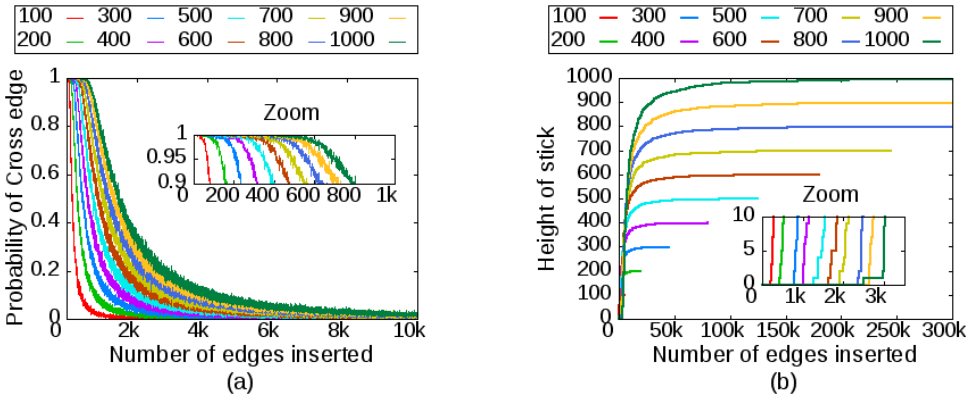

Figure 5.4: The variation of (a) p_c : Probability of next inserted edge being a cross edge, and (b) l_s : Length of broomstick, with graph density. Different lines denote different number of vertices, starting at different points.

Let l_s denote the length of the *stick* in the broomstick structure of the DFS tree. We now study the variation of l_s as the edges are inserted in the graph. Figure 5.4 (b) shows this variation of l_s for different values of n. Notice that the *stick* appears after the insertion of roughly $n \log n$ edges (see the zoomed part of Figure 5.4 (b)). After that l_s increases rapidly to reach almost 90% of its height within just $\approx 3n \log n$ edges, followed by a slow growth asymptotically approaching its maximum height only near $O(n^2)$ edges. Since any newly inserted edge with at least one endpoint on the stick necessarily becomes a back edge, the sharp decrease in p_c can be attributed to the sharp increase in l_s. We now theoretically study the reason behind the behaviour of l_s using properties of random graphs, proving explicit bounds for l_s.

5.5.2 Length of the stick

The appearance of broomstick after insertion of $n \log n$ edges as shown in Figure 5.4 (b) can be explained by the connectivity threshold for random graphs (refer to Theorem 5.1). Until the graph becomes connected (till $\Theta(n \log n)$ edges), each component hangs as a separate subtree from the pseudo root s, limiting the value of l_s to 0. To analyze the length of l_s for $m = \Omega(n \log n)$ edges, we first prove a succinct bound on the probability of existence of a long path without branching during a DFS traversal in $G(n, p)$ in the following lemma.

Lemma 5.5.1. *Given a random graph $G(n, p)$ with $p = (\log n_0 + c)/n_0$, for any integer $n_0 \leq n$ and $c \geq 1$, there exists a path without branching of length at least $n - n_0$ in the DFS tree of G with probability at least $1 - 2e^{-c}$.*

Proof. Consider any arbitrary vertex $u = x_1$, the DFS traversal starting from x_1 continues along a path without branching so long as the currently visited vertex has at least one unvisited neighbor. Let x_j denotes the j^{th} vertex visited during the DFS on $G(n, p)$ starting from x_1. The probability that x_j has at least one neighbor in the unvisited graph is $1 - (1 - p)^{n-j}$. We shall now calculate the probability that $\langle x_1, \ldots, x_{n-n_0} \rangle$ is indeed a path. Let $v = x_{n-n_0}$. We partition this sequence from v towards u into contiguous

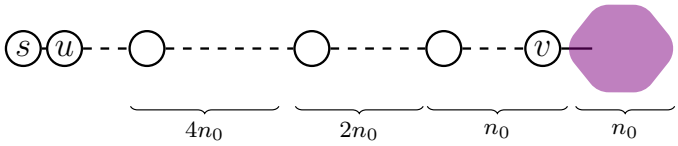

Figure 5.5: Estimating the length of 'stick' in the DFS tree.

subsequences such that the first subsequence has length n_0 and $(i+1)^{th}$ subsequence has length $2^i n_0$ (see Figure 5.5). The probability of occurrence of a path corresponding to the i^{th} subsequence is at least

$$\left(1-\left(1-\frac{\log n_0+c}{n_0}\right)^{2^i n_0}\right)^{2^i n_0} \geq \left(1-\left(\frac{1}{n_0 e^c}\right)^{2^i}\right)^{2^i n_0} \geq 1-e^{-2^i c}$$

Hence, the probability that DFS from u traverses a path of length $n-n_0$ is at least $\Pi_{i=0}^{\log_2 n}\left(1-\frac{1}{t^{2^i}}\right)$ for $t=e^c$. The value of this expression is lower bounded by $1-2e^{-c}$ using the inequality $\Pi_{i=0}^{i=\log_2 t}\left(1-\frac{1}{t^{2^i}}\right) > 1-\frac{2}{t}$, that holds for every $c \geq 1$ since it implies $t > 2$. □

In order to establish a tight bound on the length of *stick*, we need to choose the smallest value of n_0 that satisfies the following condition. Once we have a DFS path of length $n-n_0$ without branching, the subgraph induced by the remaining n_0 vertices and the last vertex of this path v (see Figure 5.5) is still connected. According to Theorem 5.1, for the graph $G(n,p)$ if the value of $p \geq \frac{1}{n_0}(\log n_0 + c)$, the subgraph on n_0 vertices will be connected with probability at least $1-e^{-c}$. Combining this observation with Lemma 5.5.1 proves that the probability that DFS tree of $G(n,p)$ is a broomstick with stick length $\geq n-n_0$ is at least $1-3e^{-c}$. This probability tends to 1 for any increasing function $c(n)$, where $c(n) \geq 1$ for all n.

Now, a graph property $\mathcal{P}$ is called a *monotone increasing* graph property if $G \in \mathcal{P}$ implies that $G+e \in \mathcal{P}$, where $G+e$ represents the graph G with an edge e added to it. Clearly, the length of the stick being at least $n-n_0$ is a monotone increasing property, as adding more edges can only increase this length. Thus, being a monotone increasing property, standard arguments[2] can be used to show that the above high probability bound for random graph $G(n,p)$ also holds for the random graph $G(n,m)$ having $m = \lceil p \cdot \binom{n}{2} \rceil$. Finally, using $c = \log n$ we get the following corollary.

Corollary 5.5.1. *For any random graph $G(n,m)$ with $m = 2^i n \log n$, its DFS tree will have bristles of size at most $n/2^i$ with probability $1-O(1/n)$.*

To demonstrate the tightness of our analysis we compare the length of the stick as predicted theoretically (for $c=1$) with the length determined experimentally in Figure 5.6, which is shown to match exactly. This phenomenon emphasizes the accuracy and tightness of our analysis.

5.5.3 Implications of broomstick property

Though the broomstick structure of DFS tree was earlier studied by Sibeyn [Sib01], the crucial difference in defining the 'stick' to be without branches proved to be extremely sig-

[2] Refer to proof of Theorem 4.1 in [FK15]

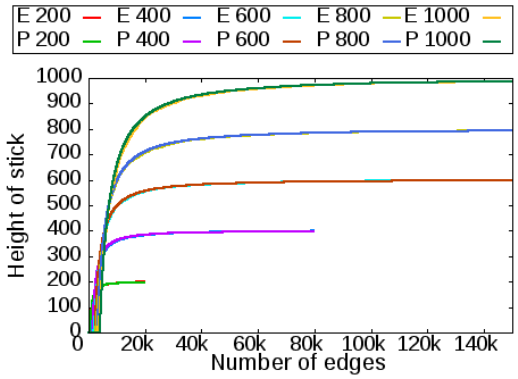

Figure 5.6: Comparison of experimentally evaluated (E) and theoretically predicted (P) value of length of the stick in the broomstick structure for different number of vertices.

nificant. To emphasize its significance we now present a few applications of the broomstick structure of DFS tree, in particular Corollary 5.5.1 to state some interesting results. Note that the absence of branches on the stick is crucial for all of the following applications.

Lemma 5.5.2. *For a uniformly random sequence of edge insertions, the number of edge insertions with both endpoints in bristles of the DFS tree will be* $O(n \log n)$

Proof. We split the sequence of edge insertions into phases and analyze the expected number of edges inserted in bristles in each phase. In the beginning of first phase there are $n \log n$ edges. In the i^{th} phase, the number of edges in the graph grow from $2^{i-1} n \log n$ to $2^i n \log n$. It follows from Corollary 5.5.1 that n_i, the size of bristles in the i^{th} phase will be at most $n/2^{i-1}$ with probability $1 - O(1/n)$. Notice that each edge inserted during i^{th} phase will choose its endpoints randomly uniformly. Therefore, in i^{th} phase the expected number of edges with both endpoints in bristles are

$$m_i = \frac{n_i^2}{n^2} m \leq 2^i n \log n / 2^{2(i-1)} = n \log n / 2^{i-2}$$

Hence, the expected number of edges inserted with both endpoints in bristles is $\sum_{i=1}^{\log n} m_i = O(n \log n)$. □

In order to rebuild the DFS tree after insertion of a cross edge, it is sufficient to rebuild only the bristles of the broomstick, leaving the *stick* intact (as cross edges cannot be incident on it). Corollary 5.5.1 describes that the size of bristles decreases rapidly as the graph becomes denser making it easier to update the DFS tree. This crucial insight is not exploited by the algorithm SDFS, SDFS-Int or WDFS. We now state the property of ADFS that exploits this insight implicitly.

Property P_2**:** ADFS modifies only the bristles of the DFS tree keeping the stick intact.

We define an incremental algorithm for maintaining a DFS for random graph to be *bristle-oriented* if executing the algorithm $\mathcal{A}$ on G is equivalent to executing the algorithm on the subgraph induced by the bristles. Clearly, ADFS is bristle-oriented owing to property P_2 and the fact that it processes only the edges with both endpoints in rerooted

subtree (refer to Section 5.2). We now state an important result for any bristle-oriented algorithm (and hence ADFS) as follows.

Lemma 5.5.3. *For any bristle-oriented algorithm $\mathcal{A}$ if the expected total time taken to insert the first $2n \log n$ edges of a random graph is $O(n^\alpha \log^\beta n)$ (where $\alpha > 0$ and $\beta \geq 0$), the expected total time taken to process any sequence of m edge insertions is $O(m + n^\alpha \log^\beta n)$.*

Proof. Recall the phases of edge insertions described in the proof of Lemma 5.5.2, where in the i^{th} phase the number of edges in the graph grow from $2^{i-1}n \log n$ to $2^i n \log n$. The size of bristles at the beginning of i^{th} phase is $n_i = n/2^{i-1}$ w.h.p.. Further, note that the size of bristles is reduced to half during the first phase, and the same happens in each subsequent phase w.h.p. (see Corollary 5.5.1). Also, the expected number of edges added to subgraph represented by the bristles in i^{th} phase is $O(n_i \log n_i)$ (recall the proof of Lemma 5.5.2). Since $\mathcal{A}$ is bristle-oriented, it will process only the subgraph induced by the bristles of size n_i in the i^{th} phase. Thus, if $\mathcal{A}$ takes $O(n^\alpha \log^\beta n)$ time in first phase, the time taken by $\mathcal{A}$ in the i^{th} phase is $O(n_i^\alpha \log^\beta n_i)$. The second term $O(m)$ comes from the fact that we would need to process each edge to check whether it lies on the stick. This can be easily done in $O(1)$ time by marking the vertices on the stick. The total time taken by $\mathcal{A}$ is $O(n^\alpha \log^\beta n)$ till the end of the first phase and in all subsequent phases is given by the following

$$
\begin{aligned}
&= m + \sum_{i=2}^{\log n} c n_i^\alpha \log^\beta n_i && \leq \sum_{i=2}^{\log n} c\Big(\frac{n}{2^{i-1}}\Big)^\alpha \log^\beta \Big(\frac{n}{2^{i-1}}\Big) \\
&\leq m + c n^\alpha \log^\beta n \sum_{i=2}^{\log n} \frac{1}{2^{(i-1)\alpha}} && \text{(for } \beta \geq 0\text{)} \\
&\leq m + c * c' n^\alpha \log^\beta n && \Big(\sum_{i=2}^{\log n} \frac{1}{2^{(i-1)\alpha}} = c', \text{for } \alpha > 0\Big)
\end{aligned}
$$

Thus, the total time taken by $\mathcal{A}$ is $O(m + n^\alpha \log^\beta n)$. □

Lemma 5.5.2 and Lemma 5.5.3 immediately implies the similarity of ADFS1 and ADFS2 as follows.

Equivalence of ADFS1 and ADFS2

On insertion of a cross edge, ADFS performs a path reversal and collects the back edges that are now converted to cross edges, to be iteratively inserted back into the graph. ADFS2 differs from ADFS1 only by imposing a restriction on the order in which these collected edges are processed. However, for sparse graphs ($m = O(n)$) this restriction does not change its worst case performance (see Table 5.1). Now, Lemma 5.5.3 states that the time taken by ADFS to incrementally process any number of edges is of the order of the time taken to process a sparse graph (with only $2n \log n$ edges). Thus, ADFS1 performs similar to ADFS2 even for dense graphs. Particularly, the time taken by ADFS1 for insertion of any $m \leq \binom{n}{2}$ edges is $O(n^2\sqrt{\log n})$, i.e., $O(n^{3/2} m_0^{1/2})$ for $m_0 = 2n \log n$. Thus, we have the following theorem.

Theorem 5.2. *Given a uniformly random sequence of arbitrary length, the expected time complexity of ADFS1 for maintaining a DFS tree incrementally is* $O(n^2\sqrt{\log n})$.

Remark: The factor of $O(\sqrt{\log n})$ in the bounds of ADFS1 and ADFS2 comes from the limitations of our analysis whereas empirically their performance matches exactly.

5.6 New algorithms for Random Graphs

Inspired by Lemma 5.5.2 and Lemma 5.5.3 we propose the following new algorithms.

Simple variant of SDFS (SDFS2) for random undirected graphs

We propose a *bristle-oriented* variant of SDFS which satisfies the properties P_1 and P_2 of ADFS, i.e., it rebuilds only the bristles of the DFS tree on insertion of only cross edges. This can be done by marking the vertices in the bristles as unvisited and performing the DFS traversal from the root of the bristles. Moreover, we also remove the non-tree edges incident on the *stick* of the DFS tree. As a result, SDFS2 would process only the edges in the bristles, making it *bristle-oriented*. Now, according to Lemma 5.5.3 the time taken by SDFS2 for insertion of $m = 2n \log n$ edges (and hence any $m \leq \binom{n}{2}$) is $O(m^2) = O(n^2 \log^2 n)$. Thus, we have the following theorem.

Theorem 5.3. *Given a random graph* $G(n, m)$*, the expected time taken by SDFS2 for maintaining a DFS tree of* G *incrementally is* $O(n^2 \log^2 n)$.

We now compare the performance of the proposed algorithm SDFS2 with the existing algorithms. Figure 5.7 (a) compares the total number of edges processed for insertion of $m = \binom{n}{2}$ edges, as a function of number of vertices in the logarithmic scale. As expected SDFS2 processes $\tilde{O}(n^2)$ edges similar to ADFS. Figure 5.7 (b) compares the number of edges processed per edge insertion as a function of number of inserted edges. Again, as expected SDFS2 performs much better than WDFS and SDFS-Int, performing asymptotically equal to ADFS as the performance differs only when the graph is very sparse ($\approx n \log n$). Interestingly, despite the huge difference in number of edges processed by SDFS2 and ADFS (see Figure 5.7 (a)), SDFS2 is faster than ADFS2 and equivalent to ADFS1 in practice (see Figure 5.15 (a)).

Experiments on directed graphs and directed acyclic graphs

The proposed algorithm SDFS2 also works for directed graphs. It is easy to show that Corollary 5.5.1 also holds for directed graphs (with different constants). Thus, the properties of broomstick structure and hence the analysis of SDFS2 can also be proved for directed graphs using similar arguments. The significance of this algorithm is highlighted by the fact that there *does not* exists any $o(m^2)$ algorithm for maintaining incremental DFS in general directed graphs. Moreover, FDFS also performs very well and satisfies the properties P_1 and P_2 (similar to ADFS in undirected graphs). Note that extension of FDFS for directed graphs is not known to have complexity $o(m^2)$, yet for random directed graphs we can prove it to be $\tilde{O}(n^2)$ using Lemma 5.5.3.

We now compare the performance of the proposed algorithm SDFS2 with the existing algorithms in the directed graphs. Figure 5.8 (a) compares the total number of edges

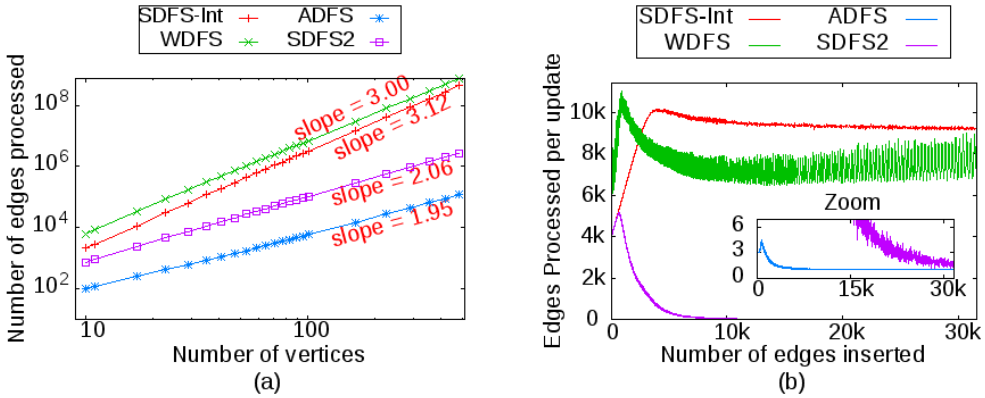

Figure 5.7: Comparison of existing and proposed algorithms on undirected graphs: (a) Total number of edges processed for insertion of $m = \binom{n}{2}$ edges for different values of n in logarithmic scale (b) Number of edges processed per edge insertion for $n = 1000$ and up to $n\sqrt{n}$ edge insertions. See Figure 5.15 for corresponding time plot.

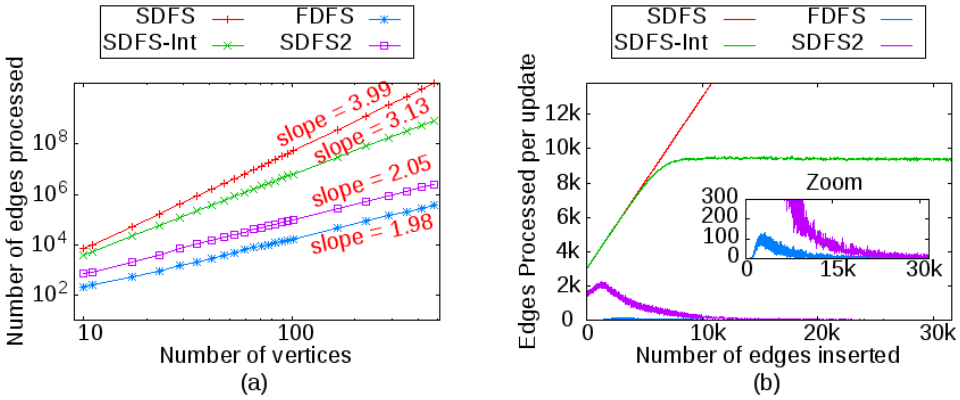

Figure 5.8: Comparison of existing and proposed algorithms on directed graphs: (a) Total number of edges processed for insertion of $m = \binom{n}{2}$ edges for different values of n in logarithmic scale (b) Number of edges processed per edge insertion for $n = 1000$ and up to $n\sqrt{n}$ edge insertions. See Figure 5.16 for corresponding time plot.

processed for insertion of $m = \binom{n}{2}$ edges, as a function of number of vertices in the logarithmic scale. As expected SDFS2 processes $\tilde{O}(n^2)$ edges similar to FDFS. Figure 5.8 (b) compares the number of edges processed per edge insertion as a function of number of inserted edges for directed graphs. Thus, the proposed SDFS2 performs much better than SDFS, and asymptotically equal to FDFS. Again despite the huge difference in number of edges processed by SDFS2 with respect to FDFS, it is equivalent to FDFS in practice (see Figure 5.8 (a) and Figure 5.16 (a)).

Finally, we compare the performance of the proposed algorithm SDFS2 with the existing algorithms in DAGs. Figure 5.9 (a) compares the total number of edges processed for insertion of $m = \binom{n}{2}$ edges, as a function of number of vertices in the logarithmic scale. Both SDFS and SDFS-Int perform equally which was not the case when the experiment was performed on undirected (Figure 5.2) or directed graphs (Figure 5.8). Moreover, SDFS2 processes around $\tilde{O}(n^3)$ edges which is more than the proven bound of $\tilde{O}(n^2)$ for

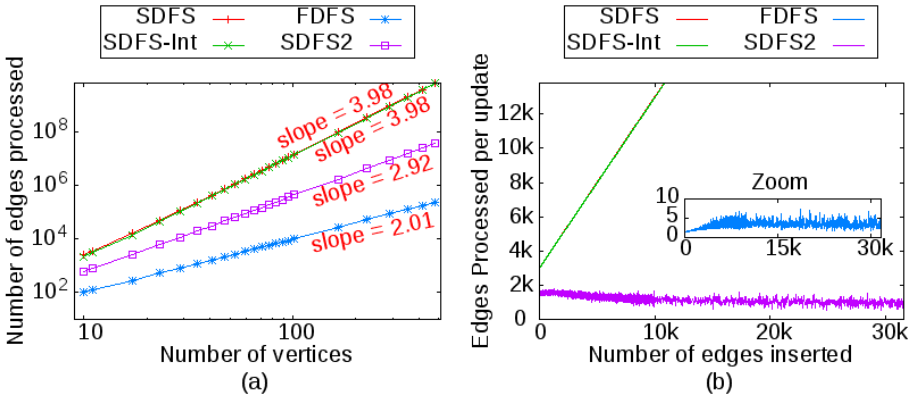

Figure 5.9: Comparison of existing and proposed algorithms on DAGs: (a) Total number of edges processed for insertion of $m = \binom{n}{2}$ edges for different values of n in logarithmic scale (b) Number of edges processed per edge insertion for $n = 1000$ and up to $n\sqrt{n}$ edge insertions. See Figure 5.17 for corresponding time plots.

undirected and directed graphs. However, FDFS processes $\tilde{O}(n^2)$ edges as expected. Figure 5.9 (b) compares the number of edges processed per edge insertion as a function of number of inserted edges. Again, both SDFS and SDFS-Int perform similarly and SDFS2 does not perform asymptotically equal to FDFS even for dense graphs. Notice that the number of edges processed by SDFS2 does not reach a peak and then asymptotically move to zero as in case of undirected and general directed graphs. Also, FDFS performs much better (similar to ADFS for undirected graphs) for DAGs as compared to directed graphs. Again, despite superior performance on random DAGs, for general DAGs the analysis of FDFS can be shown to be tight (see Section 5.8.1).

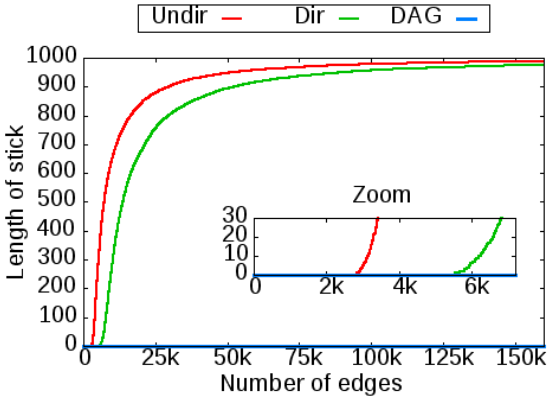

Figure 5.10: Comparison of variation of length of broomstick for 1000 vertices and different values of m. Different lines denote the variation for different type of graphs. Zoomed portion shows the start of each line.

To understand the reason behind this poor performance of SDFS-Int and SDFS2 on DAGs, we compare the variation in length of broomstick for the undirected graphs, general directed graphs and DAGs in Figure 5.10. The length of the broomstick varies as expected for undirected and general directed graphs but always remains zero for DAGs. This is be-

cause the stick will appear only if the first neighbor of the pseudo root s visited by the algorithm is the first vertex (say v_1) in the topological ordering of the graph. Otherwise v_1 hangs as a separate child of s because it not reachable from any other vertex in the graph. Since the edges in $G(n, m)$ model are permuted randomly, with high probability v_1 may not be the first vertex to get connected to s. The same argument can be used to prove branchings at every vertex on the stick. Hence, with high probability there would be some bristles even on the pseudo root s. This explains why SDFS-Int performs equal to SDFS as it works same as SDFS until all the vertices are visited. SDFS2 only benefits from the inserted edges being reverse cross edges which are valid in a DFS tree and hence avoids rebuilding on every edge insertion. Thus, Corollary 5.5.1 and hence the bounds for SDFS2 proved in Theorem 5.3 are not valid for the case of DAGs as resulting in performance described above. Moreover, the absence of the broomstick phenomenon can also be proved for other models of random graphs for DAGs [CMP+10] using the same arguments.

Finally, Lemma 5.5.2 also inspires the following interesting applications of SDFS2 in the semi-streaming environment as follows.

Semi-streaming algorithms

In the streaming model we have two additional constraints. Firstly, the input data can be accessed only sequentially in the form of a stream. The algorithm can do multiple passes on the stream, but cannot access the entire stream. Secondly, the working memory is considerably smaller than the size of the entire input stream. For graph algorithms, a semi-streaming model allows the size of the working memory to be $\tilde{O}(n)$.

The DFS tree can be trivially computed using $O(n)$ passes over the input graph in the semi-streaming environment, each pass adding one vertex to the DFS tree. However, computing the DFS tree in even $\tilde{O}(1)$ passes is considered hard [FHLT15]. To the best of our knowledge, it remains an open problem to compute the DFS tree using even $o(n)$ passes in any relaxed streaming environment [O'C09, Ruh03]. Now, some of the direct applications of a DFS tree in undirected graphs are answering connectivity, biconnectivity and 2-edge connectivity queries. All these problems are addressed efficiently in the semi-streaming environment using a single pass by the classical work of Westbrook and Tarjan [WT92]. On the other hand, for the applications of a DFS tree in directed graphs as strong connectivity, strong lower bounds of space for single-pass semi-streaming algorithms have been shown . Borradaile et al. [BMM14] showed that any algorithm requires a a working memory of $\Omega(\epsilon m)$ to answer queries of strong connectivity, acyclicity or reachability from a vertex require with probability greater than $(1 + \epsilon)/2$.

We now propose a semi-streaming algorithm for maintaining Incremental DFS for random graphs. The key idea to limit the storage space required by this algorithm is to just discard those edges from the stream whose at least one endpoint is on the stick of the DFS tree. As described earlier, this part of DFS tree corresponding to the stick will never be modified by the insertion of any edge. If both the endpoints of the edge lie in bristles, we update the DFS tree using ADFS/SDFS2. Lemma 5.5.2 implies that the expected number of edges stored will be $O(n \log n)$. In case we use SDFS2 (for directed graphs) we also delete the non-tree edges incident on the *stick*. Hence, we have the following theorem.

Theorem 5.4. *Given a random graph $G(n, m)$, there exists a single pass semi-streaming algorithm for maintaining the DFS tree incrementally, that requires $O(n \log n)$ space.*

Further, for random graphs even strong connectivity can be solved using a single pass in the streaming environment by SDFS2 as follows. Now, SDFS2 keeps only the tree edges and the edges in the bristles. For answering strong connectivity queries, we additionally store the highest edge from each vertex on the stick. The strongly connected components can thus be found by a single traversal on the DFS tree [Tar72]. Thus, our semi-streaming algorithm SDFS2 not only gives a solution for strong connectivity in the streaming setting but also establishes the difference in its hardness for general graphs and random graphs. To the best of our knowledge no such result was known for any graph problem in streaming environment prior to our work. Thus, we have the following theorem.

Theorem 5.5. *Given a random graph $G(n, m)$, there exists a single pass semi-streaming algorithm for maintaining a data structure that answers strong connectivity queries in G incrementally, requiring $O(n \log n)$ space.*

5.7 Incremental DFS on real graphs

We now evaluate the performance of existing and proposed algorithms on real graphs. Recall that for random graphs, bristles represent the entire DFS tree until the insertion of $\Theta(n \log n)$ edges. This forces SDFS2 to rebuild the whole tree requiring total $\Omega(n^2)$ time even for sparse random graphs, whereas ADFS and FDFS only partially rebuild the DFS tree and turn out to be much better for sparse random graphs (see Figure 5.7 (b), 5.8 (b) and 5.9 (b)). Now, most graphs that exist in real world are known to be sparse [Mel06]. Here again, both ADFS and FDFS perform much better as compared to SDFS2 and other existing algorithms. Thus, we propose another simple variant of SDFS (SDFS3), which is both easy to implement and performs very well even on real graphs (much better than SDFS2).

5.7.1 Proposed algorithms for real graphs (SDFS3)

The primary reason behind the superior performance of ADFS and FDFS is the partial rebuilding of the DFS tree upon insertion of an edge. However, the partial rebuilding by SDFS2 is significant only when the broomstick has an appreciable size, which does not happen until the very end in most of the real graphs. With this insight, we propose new algorithms for directed and undirected graphs. The aim is to rebuild only that region of the DFS tree which is affected by the edge insertion.

- **Undirected Graphs**
  On insertion of a cross edge (x, y), ADFS rebuilds one of the two *candidate* subtrees hanging from $LCA(x, y)$ containing x or y. We propose algorithm SDFS3 that will rebuild only the smaller subtree (less number of vertices) among the two candidate subtrees (say $x \in T_1$ and $y \in T_2$). This heuristic is found to be extremely efficient compared to rebuilding one of T_1 or T_2 arbitrarily. The smaller subtree, say T_2, can be identified efficiently by simultaneous traversal in both T_1 and T_2. and terminate as soon as either one is completely traversed. This takes time of the order of $|T_2|$. We then mark the vertices of T_2 as unvisited and start the traversal from y in T_2, hanging the newly created subtree from edge (x, y).

- **Directed Graphs**
  On insertion of an anti-cross edge (x, y), FDFS rebuilds the vertices reachable from y in the subgraph induced by a *candidate set* of subtrees described in Section 5.2. FDFS identifies this affected subgraph using the DFN number of the vertices. Thus, this DFN number also needs to be updated separately after rebuilding the DFS tree. This is done by building an additional data structure while the traversal is performed, which aids in updating the DFN numbers efficiently. We propose SDFS3 to simply mark all the subtrees in this *candidate set* as unvisited and proceed the traversal from (x, y). The traversal then continues from each unvisited root of the subtrees marked earlier, implicitly restoring the DFN number of each vertex.

5.7.2 Experimental Setup

The algorithms are implemented in C++ using STL (standard template library), and built with GNU g++ compiler (version 4.4.7) with optimization flag $-O3$. The correctness of our code was exhaustively verified on random inputs by ensuring the absence of anti-cross edges (or cross edge) in directed (or undirected) graphs. Our experiments were run on Intel Xeon E5-2670V 2.5 GHz 2 CPU-IvyBridge (20-cores per node) on HP-Proliant-SL-230s-Gen8 servers with 1333 MHz DDR3 RAM of size 768 GB per node. Each experiment was performed using a single dedicated processor.

5.7.3 Datasets used for evaluation on Real Graphs

We consider the following types of graphs in our experiments:

- **Internet topology:** These datasets represent snapshots of network topology on CAIDA project (*asCaida* [LKF05, LK14]), Oregon Route Views Project's Autonomous Systems (*ass733* [LKF05, LK14]) and Internet autonomous systems (*intTop* [ZLMZ05, Kun16]).

- **Collaboration networks:** These datasets represent the collaboration networks as recorded on arXiv's High-Energy-Physics groups of Phenomenology (*arxvPh* [LKF07, DH14, Kun16]) and Theory (*arxvTh* [LKF07, DH14, Kun16]), and on DBLP (*dblp* [Ley02, DH14, Kun16]).

- **Online communication:** These datasets represent communication of linux kernel messages (*lnKMsg* [Kun16]), Gnutella peer-to-peer file sharing network (*gnutella* [RIF02, LK14]), Slashdot's message exchange (*slashDt* [GKL08, Kun16]), Facebook's wall posts (*fbWall* [VMCG09, Kun16]), Democratic National Committee's (DNC) email correspondence (*dncCoR* [Kun16]), Enron email exchange (*enron* [KY04, Kun16]), Digg's reply correspondence (*digg* [CSJS09, Kun16]) and UC Irvine message exchange (*ucIrv* [OP09, Kun16])

- **Friendship networks:** These datasets represent the friendship networks of Flickr (*flickr* [MKG$^+$08, Kun16], Digg (*diggNw* [HL12, Kun16]), Epinion (*epinion* [MA05, Kun16]), Facebook (*fbFrnd* [VMCG09, Kun16]) and Youtube (*youTb* [Mis09, Kun16]).

- **Other interactions:** These datasets represent the other networks as Chess game interactions (*chess* [Kun16]), user loans on Prosper (*perLoan* [Kun16]), hyperlink

network of Wikipedia (*wikiHy* [Mis09, Kun16]), voting in elections on Wikipedia (*wikiEl* [LHK10, Kun16]) and conflict resolution on Wikipedia (*wikiC* [BKLvR09, Kun16]).

In some of these datasets there are some rare instances in which edges are deleted (not present in new snapshot). Thus, in order to use these datasets for evaluation of incremental algorithms we ignore the deletion of these edges (and hence reinsertion of deleted edges). Moreover, in several datasets the edges are inserted in form of batches (having same insertion time), where the number of batches are significantly lesser than the number of inserted edges. Almost all the algorithms (except FDFS and SDFS3) can be tweaked to handle such batch insertions more efficiently, updating the DFS tree once after insertion of an entire batch, instead of treating every edge insertion individually.

Dataset	n	$m \mid m^*$	$\frac{m}{n} \mid \frac{m}{m^*}$	ADFS1	ADFS2	SDFS3	SDFS2	SDFS	WDFS
ass733	7.72K	21.47K	2.78	**1.00**	1.88	34.12	639.50	1.13K	2.99K
		721.00	29.77	**1.00**	2.71	38.43	35.57	54.14	95.43
intTop	34.76K	107.72K	3.10	**1.00**	2.14	111.32	3.78K	8.15K	14.65K
		18.27K	5.89	**1.00**	6.07	99.47	320.49	1.83K	2.24K
fbFrnd	63.73K	817.03K	12.82	**1.00**	2.18	146.58	2.02K	14.67K	11.75K
		333.92K	2.45	**1.00**	8.10	141.07	491.24	7.63K	4.27K
wikiC	116.84K	2.03M	17.36	**1.00**	1.82	249.45	3.09K	>22.56K	>22.56K
		205.59K	9.86	**1.00**	2.26	246.49	2.69K	4.39K	3.35K
arxvTh	22.91K	2.44M	106.72	**1.00**	1.81	28.31	3.41K	>39.96K	9.72K
		210.00	11.64K	**1.00**	6.74	32.01	8.63	13.24	2.84K
arxvPh	28.09K	3.15M	112.07	**1.00**	2.38	57.94	2.54K	>36.29K	11.32K
		2.26K	1.39K	**1.00**	8.25	70.75	103.23	192.22	3.17K
dblp	1.28M	3.32M	2.59	**1.00**	1.60	>22.07K	>22.07K	>22.07K	>22.07K
		1.72M	1.93	**1.00**	1.84	>21.26K	>21.26K	>21.26K	>21.26K
youTb	3.22M	9.38M	2.91	**1.00**	3.53	>347.00	>347.00	>347.00	>347.00
		203.00	46.18K	1.26	2.26	>322.18	1.00	**1.00**	260.73

Table 5.2: Comparison of time taken by different algorithms, relative to the fastest (shown in bold), for maintaining incremental DFS on real undirected graphs. See Table 5.4 for corresponding table comparing the exact performance of different algorithms.

5.7.4 Evaluation

The comparison of the performance of the existing and the proposed algorithms for real undirected graphs and real directed graphs is shown in Table 5.2 and Table 5.3 respectively. To highlight the relative performance of different algorithms, we present the time taken by them relative to that of the fastest algorithm (see Section 5.10 for the exact time and memory used by different algorithms). In case the time exceeded 100hrs the process was terminated, and we show the relative time in the table with a '>' sign and the ratio corresponding to 100hrs. For each dataset, the first row corresponds to the experiments in which the inserted edges are processed one by one, and the second row corresponds to the experiments in which the inserted edges are processed in batches (m^* denotes the

Dataset	n	$m \mid m^*$	$\frac{m}{n} \mid \frac{m}{m^*}$	FDFS	SDFS3	SDFS2	SDFS
dncCoR	1.89K	5.52K	2.92	1.55	**1.00**	2.27	9.86
		4.01K	1.38	1.55	**1.00**	2.00	7.18
ucIrv	1.90K	20.30K	10.69	1.69	**1.00**	2.25	21.81
		20.12K	1.01	1.78	**1.00**	2.35	22.14
chess	7.30K	60.05K	8.22	1.94	**1.00**	2.54	20.00
		100.00	600.46	52.04	26.14	**1.00**	**1.00**
diggNw	30.40K	85.25K	2.80	**1.00**	1.33	3.60	14.50
		81.77K	1.04	**1.00**	1.38	3.78	11.96
asCaida	31.30K	97.84K	3.13	**1.00**	4.31	13.60	64.71
		122.00	801.98	12.57	42.62	1.01	**1.00**
wikiEl	7.12K	103.62K	14.55	1.01	**1.00**	2.58	51.80
		97.98K	1.06	1.00	**1.00**	2.53	52.38
slashDt	51.08K	130.37K	2.55	1.03	**1.00**	2.78	5.85
		84.33K	1.55	1.04	**1.00**	2.07	3.79
lnKMsg	27.93K	237.13K	8.49	1.82	**1.00**	2.40	23.24
		217.99K	1.09	1.77	**1.00**	2.30	23.13
fbWall	46.95K	264.00K	5.62	1.29	**1.00**	2.49	14.84
		263.12K	1.00	1.31	**1.00**	2.73	17.11
enron	87.27K	320.15K	3.67	**1.00**	1.55	5.66	67.58
		73.87K	4.33	**1.00**	1.48	2.61	14.00
gnutella	62.59K	501.75K	8.02	1.23	**1.00**	2.54	19.13
		9.00	55.75K	1.17K	1.04K	1.03	**1.00**
epinion	131.83K	840.80K	6.38	1.32	**1.00**	2.29	17.77
		939.00	895.42	95.27	93.62	**1.00**	1.00
digg	279.63K	1.73M	6.19	**1.00**	1.18	3.96	>29.28
		1.64M	1.05	**1.00**	1.34	4.08	>30.92
perLoan	89.27K	3.33M	37.31	**1.00**	7.10	30.70	>639.03
		1.26K	2.65K	2.13	13.18	**1.00**	1.01
flickr	2.30M	33.14M	14.39	-	-	-	-
		134.00	247.31K	>476.50	>476.50	1.01	**1.00**
wikiHy	1.87M	39.95M	21.36	-	-	-	-
		2.20K	18.18K	>69.26	>69.26	**1.00**	1.13

Table 5.3: Comparison of time taken by different algorithms, relative to the fastest (shown in bold), for maintaining incremental DFS on real directed graphs. If all algorithms exceed 100hrs giving no fastest algorithm, their corresponding relative time is not shown (-). See Table 5.5 for corresponding table comparing the exact performance of different algorithms.

corresponding number of batches). The density of a graph can be judged by comparing the average degree (m/n) with the number of vertices (n). Similarly, the batch density of a graph can be judged by comparing the average size of a batch (m/m^*) with the number of vertices (n).

For undirected graphs, Table 5.2 clearly shows that ADFS1 outperforms all the other algorithms irrespective of whether the edges are processed one by one or in batches (except *youTb*). Moreover, despite ADFS2 having better worst case bounds than ADFS1, the overhead of maintaining its data structure $\mathcal{D}$ leads to inferior performance as com-

pared to ADFS1. Also, SDFS2 significantly improves over SDFS (> 2 times). However, by adding a simple heuristic, SDFS3 improves over SDFS2 by a huge margin (> 10 times) which becomes even more significant when the graph is very dense (*arxvTh* and *arxvPh*). Also, note that even SDFS3 performs a lot worse than ADFS (> 30 times) despite having a profound improvement over SDFS2. Further, despite having good worst case bounds, WDFS seems to be only of theoretical interest and performs worse than even SDFS in general. However, if the graph is significantly dense (*fbFrnd*, *wikiC*, *arxvTh* and *arxvPh*), WDFS performs better than SDFS but still far worse than SDFS2. Now, in case of batch updates, SDFS3 is the only algorithm that is unable to exploit the insertion of edges in batches. Hence, SDFS3 performs worse than SDFS2 and even SDFS if the batch density is significantly high (*arxvTh* and *youTb*). Finally, if the batch density is extremely high (*youTb*), the simplicity of SDFS and SDFS2 results in a much better performance than even ADFS.

Observations: For real undirected graphs

- ADFS outperforms all other algorithms by a huge margin, with ADFS1 always performing mildly better than ADFS2.
- SDFS2 mildly improves SDFS, whereas SDFS3 significantly improves SDFS2.
- WDFS performs even worse than SDFS for sparse graphs.

For directed graphs, Table 5.3 shows that both FDFS and SDFS3 perform almost equally well (except *perLoan*) and outperform all other algorithms when the edges are processed one by one. In general SDFS3 outperforms FDFS marginally when the graph is dense (except *slashDt* and *perLoan*). The significance of SDFS3 is further highlighted by the fact that it is much simpler to implement as compared to FDFS. Again, SDFS2 significantly improves over SDFS (> 2 times). Further, by adding a simple heuristic, SDFS3 improves over SDFS2 (> 2 times), and this improvement becomes more pronounced when the graph is very dense (*perLoan*). Now, in case of batch updates, both FDFS and SDFS3 are unable to exploit the insertion of edges in batches. Hence, they perform worse than SDFS and SDFS2 for batch updates, if the average size of a batch is at least 600. SDFS and SDFS2 perform almost equally well in such cases with SDFS performing marginally better than SDFS2 when the batch density is significantly high (*asCaida*, *gnutella* and *flickr*).

Observations: For real directed graphs

- FDFS and SDFS3 outperform all other algorithms unless batch density is high, where trivial SDFS is better.
- SDFS3 performs better than FDFS in dense graphs.
- SDFS2 mildly improves SDFS, and SDFS3 mildly improves SDFS2.

Overall, we propose the use of ADFS1 and SDFS3 for undirected and directed graphs respectively. Although SDFS3 performs very well on real graphs, its worst case time complexity is no better than that of SDFS on general graphs (see Section 5.8.2). Finally, in

case the batch density of the input graph is substantially high, we can simply use the trivial SDFS algorithm.

Remark: The improvement of SDFS3 over SDFS2 is substantially better on undirected graphs than on directed graphs. Even then ADFS1 outperforms SDFS3 by a huge margin. Also, when the batch density is extremely high ($youTb$), ADFS1 performs only mildly slower than the fastest algorithm (SDFS). These observations further highlight the significance of ADFS1 in practice.

5.8 Tightness of Worst case bounds

We now present worst case inputs to demonstrate the worst case bounds for the empirically efficient algorithms, FDFS and SDFS3.

5.8.1 Worst Case Input for FDFS

We now describe a sequence of $O(m)$ edge insertions in a directed acyclic graph for which FDFS takes $\Theta(mn)$ time to maintain DFS tree. Consider a directed acyclic graph $G = (V, E)$ where the set of vertices V is divided into two sets $A = \{a_1, a_2, ..., a_{n/2}\}$ and $B = \{b_1, b_2, ..., b_{n/2}\}$, each of size $n/2$. The vertices in both A and B are connected in the form of a chain (see Figure 5.11 (a), which is the DFS tree of the graph). Additionally, set of vertices in B are connected using $m - n/2$ edges (avoiding cycles), i.e. there can exist edges between b_i and b_j, where $i < j$. For any $n \leq m \leq \binom{n}{2}$, we can add $\Theta(m)$ edges to B as described above. Now, we add $n/2$ more edges as described below.

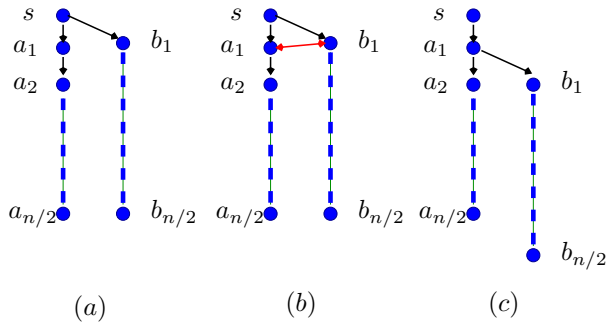

Figure 5.11: Example to demonstrate the tightness of analysis of FDFS. (a) Initial DFS tree of the graph G. (b) Insertion of a cross edge (a_1, b_1). (c) The resultant DFS tree.

We first add the edge (a_1, b_1) as shown in Figure 5.11 (b). On addition of an edge (x, y), FDFS processes all outgoing edges of the vertices having rank $\phi(x) < \phi' \leq \phi(y)$, where ϕ is the post order numbering of the DFS tree. Clearly, the set of such vertices is the set B. Hence, all the $\Theta(m)$ edges in B will be processed to form the final DFS tree as shown in Figure 2.3 (c). We next add the edge (a_2, b_1) which will again lead to processing of all edges in B, and so on. This process can be repeated $n/2$ times adding each (a_i, b_1), for $i = 1, 2, ..., n/2$ iteratively. Thus, for $n/2$ edge insertions, FDFS processes $\Theta(m)$ edges each, requiring a total of $\Theta(mn)$ time to maintain the DFS tree. Hence, overall time

required for insertion of m edges is $\Theta(mn)$, as FDFS has a worst case bound of $O(mn)$. Thus, we have the following theorem.

Theorem 5.6. *For each value of $n \leq m \leq \binom{n}{2}$, there exists a sequence of m edge insertions for which FDFS requires $\Theta(mn)$ time to maintain the DFS tree.*

5.8.2 Worst Case Input for SDFS3

We now describe a sequence of m edge insertions for which SDFS3 takes $\Theta(m^2)$ time. Note that since SDFS3 is necessarily an improvement over SDFS2 and hence even SDFS, the worst case example would also serve for SDFS2 and SDFS. Consider a graph $G = (V, E)$ where the set of vertices V is divided into two sets V' and I, each of size $\Theta(n)$. The vertices in V' are connected in the form of a three chains (see Figure 5.12 (a)) and the vertices in I are isolated vertices. Thus, it is sufficient to describe only the maintenance of DFS tree for the vertices in set V', as the vertices in I will exist as isolated vertices connected to the dummy vertex s in the DFS tree (recall that s is the root of the DFS tree).

We divide the sequence of edge insertions into k *phases*, where each phase is further divided into k *stages*. At the beginning of each *phase*, we identify three chains having vertex sets from the set V', namely $A = \{a_1, ..., a_k\}$, $X = \{x_1, ..., x_p\}$ in the first chain, $B = \{b_1, ..., b_l\}$ and $Y = \{y_1, ..., y_q\}$ in the second chain and $C = \{c_1, ..., c_k\}$, $Z = \{z_1, ..., z_r\}$ in the third chain as shown in Figure 5.12 (a). The constants $k, p, q, r = \Theta(\sqrt{m})$ such that $q > r + k$ and $p \approx q + r + k$. We then add $e_Z = \Theta(m)$ edges to the set Z, $e_y = e_z + k + 1$ edges to Y and $e_x = e_z + e_y$ edges to X, which is overall still $\Theta(m)$. The size of A and C is k in the first phase and decreases by 1 in each the subsequent phases. Figure 5.12 (a) shows the DFS tree of the initial graph.

Now, the *first stage* of the phase starts with addition of the cross edge (b_1, c_1) as shown in Figure 5.12 (b). Clearly, s is the LCA of the inserted edge and SDFS3 would rebuild the smaller of the two subtrees $T(b_1)$ and $T(c_1)$. Since $q > r$, SDFS3 would hang $T(c_1)$ through edge (b_1, c_1) and perform partial DFS on $T(c_1)$ requiring to process $\Theta(m)$ edges in Z. This completes the first stage with the resultant DFS tree shown in the Figure 5.12 (c). This process continues for k stages, where in i^{th} stage, $T(c_1)$ would initially hang from b_{i-1} and (b_i, c_1) would be inserted. The DFS tree at the end of k^{th} stage is shown in Figure 5.12 (d). At the end of k stages, every vertex in B is connected to the vertex c_1, hence we remove it from C for the next phase. For this we first add the edge (a_1, c_1). Since both $T(b_1)$ and $T(a_1)$ have approximately same number of vertices (as $p \approx q+r+k$), we add constant number of vertices (if required) to Z from I to ensure $T(b_1)$ is rebuilt. The resultant DFS tree is shown in Figure 5.12 (e). Finally, we add (a_2, c_1). Again both $T(c_1)$ and $T(a_2)$ have approximately same number of vertices, so we add constant number of vertices from I to X ensuring $T(a_2)$ is rebuild as shown in Figure 5.12 (f). Note the similarity between Figures 5.12 (a) and 5.12 (f). In the next phase, the only difference is that $A' = \{a_2, ..., a_k\}$, $C' = \{c_2, ..., c_k\}$ and $s' = c_1$. In each phase one vertex each from A and C are removed and constant number of vertices from I are removed. Hence the phase can be repeated k times.

Thus, we have k phases each having k stages. Further, in each stage we add a single cross edge forcing SDFS3 to process $\Theta(m)$ edges to rebuild the DFS tree. Thus, the total number of edges added to the graph is $k * k = \Theta(m)$ and the total time taken by ADFS1 is $k * k * \Theta(m) = \Theta(m^2)$. Hence, we get the following theorem for any $n \leq m \leq \binom{n}{2}$.

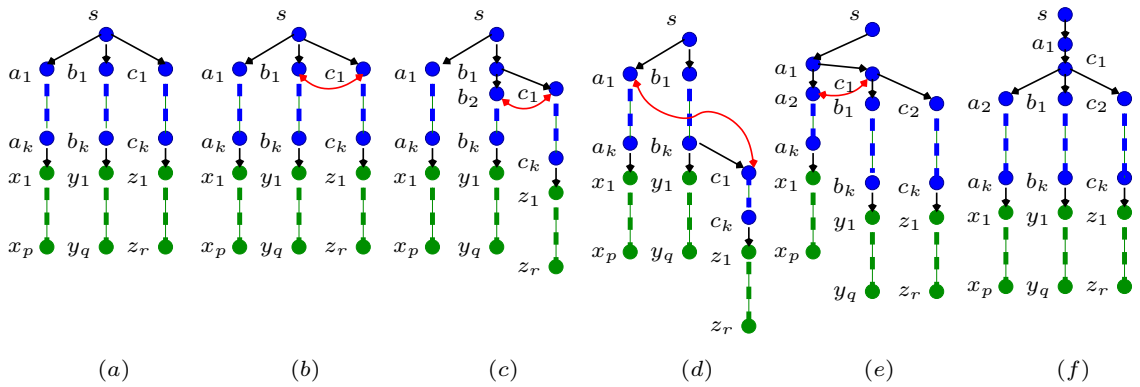

Figure 5.12: Example to demonstrate the tightness of the SDFS3. (a) Beginning of a phase with vertex sets A, B and X. (b) Phase begins with addition of two vertex sets C and D. The first stage begins by inserting a back edge (a_1, b_k) and a cross edge (b_1, c_k). (c) The rerooted subtree with the edges in $A \times X$ and (b_k, a_1) as cross edges. (d) Final DFS tree after the first stage. (e) Final DFS tree after first phase. (f) New vertex sets A', B' and X for the next phase.

Theorem 5.7. *For each value of* $n \leq m \leq \binom{n}{2}$*, there exists a sequence of* m *edge insertions for which SDFS3 requires* $\Theta(m^2)$ *time to maintain the DFS tree.*

Remark: The worst case example mentioned above (say G_1) would also work without X, Y and Z. Consider a second example (say G_2), where we take size of $A = 2 * k + 2$,$B = k + 1$ and $C = k$ and the vertices of C have $\Theta(m)$ edges amongst each other. The same sequence of edge insertions would also force SDFS3 to process $\Theta(m^2)$ edges. However, G_1 also ensures the same worst case bound for SDFS3 if it chooses the subtree with lesser edges instead of the subtree with lesser vertices, which is an obvious workaround of the example G_2. The number of edges e_x, e_y and e_z are chosen precisely to counter that argument.

5.9 Time Plots for experiments

In this section we present the corresponding time plots for experiments performed earlier which were measured in terms of number of edges processed. The comparison of the existing incremental algorithms for random undirected graphs are shown in Figure 5.13 and Figure 5.14. The comparison of the existing and proposed algorithms for random undirected graphs, random directed graphs and random DAGs are shown in Figure 5.15, Figure 5.16 and Figure 5.17 respectively.

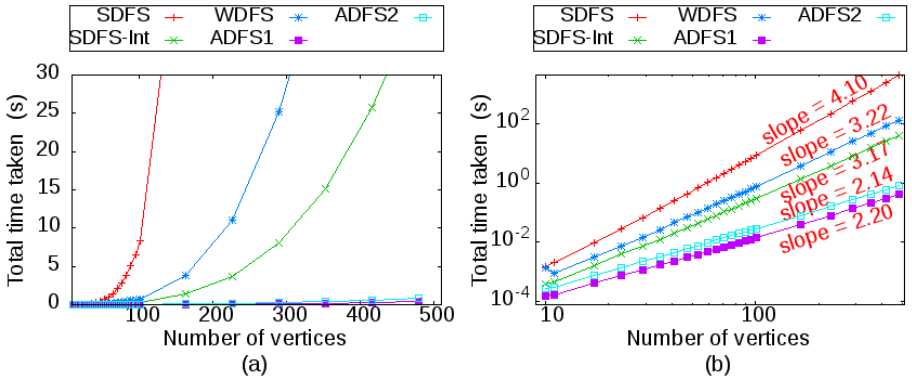

Figure 5.13: Total time taken by existing algorithms for insertion of $m = \binom{n}{2}$ edges for different values of n. (a) Normal scale. (b) Logarithmic scale.

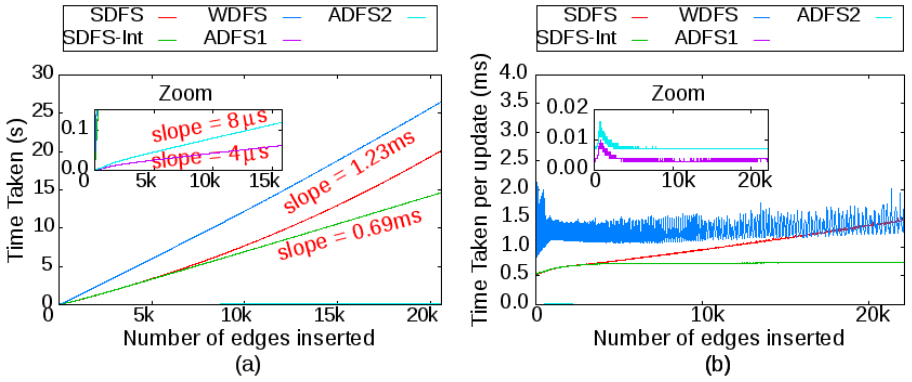

Figure 5.14: For $n = 1000$ and up to $n\sqrt{n}$ edge insertions the plot shows (a) Total time taken, (b) Time taken per edge insertion, by the existing algorithms.

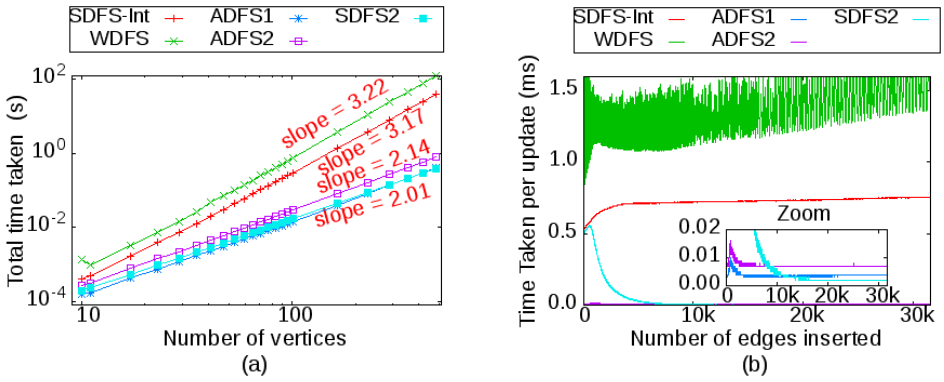

Figure 5.15: Comparison of existing and proposed algorithms on undirected graphs: (a) Total time taken for insertion of $m = \binom{n}{2}$ edges for different values of n. (b) Time taken per edge insertion for $n = 1000$ and up to $n\sqrt{n}$ edge insertions.

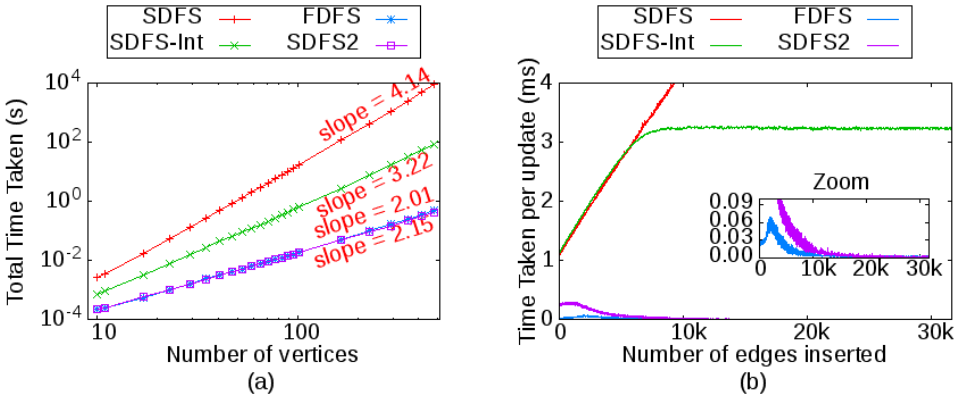

Figure 5.16: Comparison of existing and proposed algorithms on directed graphs: (a) Total time taken for insertion of $m = \binom{n}{2}$ edges for different values of n in logarithmic scale (b) Time taken per edge insertion for $n = 1000$ and up to $n\sqrt{n}$ edge insertions.

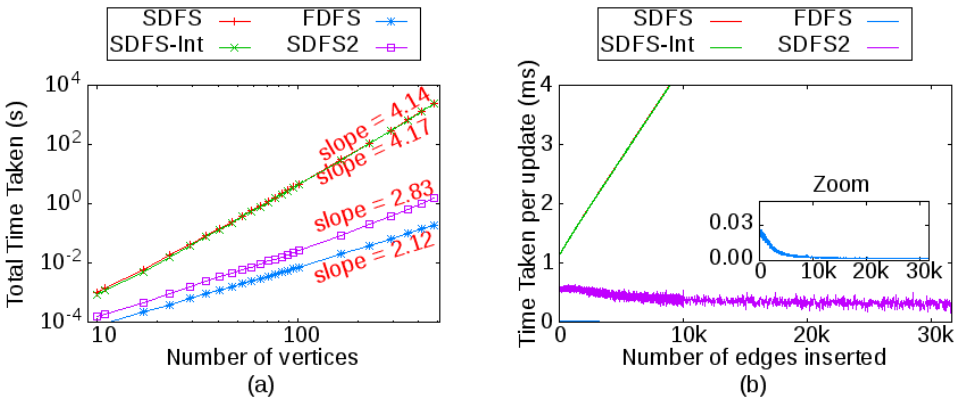

Figure 5.17: Comparison of existing and proposed algorithms on DAGs: (a) Total time taken for insertion of $m = \binom{n}{2}$ edges for different values of n in logarithmic scale (b) Time taken per edge insertion for $n = 1000$ and up to $n\sqrt{n}$ edge insertions.

5.10 Exact performance comparison for real graphs

The performance of different algorithms in terms of time and memory required on real undirected graphs and real directed graphs is shown in Table 5.4 and Table 5.5 respectively.

5.11 Discussion

Our experimental study of existing algorithms for incremental DFS on random graphs presented some interesting inferences. Upon further investigation, we discovered an important property of the structure of DFS tree in random graphs: the broomstick structure. We then theoretically proved the variation in length of the *stick* of the DFS tree as the graph density increases, which also exactly matched the experimental results. This led to several interesting applications, including the design of an extremely simple algorithm SDFS2. This algorithm theoretically matches and experimentally outperforms the state-of-the-art algorithm in dense random graphs. It can also be used as a single pass semi-streaming

algorithm for incremental DFS as well as strong connectivity in random graphs, which also establishes the difference in hardness of strong connectivity in general graphs and random graphs. Finally, for real world graphs, which are usually sparse, we propose a new simple algorithm SDFS3 which performs much better than SDFS2. Despite being extremely simple, it almost always matches the performance of FDFS in directed graphs. However, for undirected graphs ADFS was found to outperform all algorithms (including SDFS3) by a huge margin motivating its use in practice.

For future research directions, recall that ADFS (see Inference I_2) performs extremely well even on sparse random graphs. Similarly, the performance of FDFS and SDFS3 is also very good even on sparse random graphs. However, none of these have asymptotic bounds any better than $\tilde{O}(n^2)$. After preliminary investigation, we believe that the asymptotic bounds for ADFS and FDFS (in DAGs) should be $O(m + n\text{polylog}n)$ for random graphs. Also, we believe that the asymptotic bounds for SDFS3 and FDFS (in directed graphs) should be $O(m + n^{4/3}\text{polylog}n)$ for random graphs. It would be interesting to see if it is possible to prove these bounds theoretically.

Dataset	n	$m \vert m^*$	$\frac{m}{n} \vert \frac{m}{m^*}$	ADFS1		ADFS2		SDFS3		SDFS2		SDFS		WDFS	
ass733	7.72K	21.47K	2.78	**0.08s**	35.09M	0.15s	37.17M	2.73s	39.91M	51.16s	**33.25M**	1.51m	**33.25M**	3.99m	450.06M
		721.00	29.77	**0.07s**	35.11M	0.19s	36.66M	2.69s	**31.98M**	2.49s	32.77M	3.79s	32.77M	6.68s	385.61M
intTop	34.76K	107.72K	3.10	**0.50s**	160.14M	1.07s	162.58M	55.66s	150.17M	31.50m	152.12M	1.13h	**148.95M**	2.04h	2.57G
		18.27K	5.89	**0.55s**	160.80M	3.34s	169.44M	54.71s	150.84M	2.94m	151.81M	16.76m	**146.00M**	20.51m	2.56G
fbFrnd	63.73K	817.03K	12.82	**10.26s**	817.67M	22.39s	837.23M	25.06m	**725.08M**	5.75h	747.55M	41.80h	748.66M	33.48h	15.13G
		333.92K	2.45	**10.46s**	816.72M	1.41m	844.47M	24.59m	**724.12M**	1.43h	745.77M	22.17h	746.42M	12.40h	15.14G
wikiC	116.84K	2.03M	17.36	**15.96s**	1.89G	29.01s	1.91G	1.11h	**1.65G**	13.71h	1.67G	>100.00h	-	>100.00h	-
		205.59K	9.86	**15.78s**	1.89G	35.67s	1.91G	1.08h	**1.65G**	11.77h	1.67G	19.23h	1.67G	14.68h	38.50G
arxvTh	22.91K	2.44M	106.72	**9.01s**	2.10G	16.28s	2.11G	4.25m	1.81G	8.53h	**1.81G**	>100.00h	-	24.33h	39.48G
		210.00	11.64K	**8.04s**	2.61G	54.22s	2.77G	4.29m	2.34G	1.16m	**2.33G**	1.77m	2.33G	6.34h	3.94G
arxvPh	28.09K	3.15M	112.07	**9.92s**	2.69G	23.59s	2.71G	9.58m	2.33G	7.00h	**2.32G**	>100.00h	-	31.19h	50.80G
		2.26K	1.39K	**8.15s**	2.69G	1.12m	2.70G	9.61m	2.33G	14.02m	**2.32G**	26.11m	**2.32G**	7.18h	26.12G
dblp	1.28M	3.32M	2.59	**16.31s**	**4.51G**	26.12s	5.14G	>100.00h	-	>100.00h	-	>100.00h	-	>100.00h	-
		1.72M	1.93	**16.93s**	**4.51G**	31.17s	4.76G	>100.00h	-	>100.00h	-	>100.00h	-	>100.00h	-
youTb	3.22M	9.38M	2.91	**17.29m**	**13.29G**	1.02h	14.04G	>100.00h	-	>100.00h	-	>100.00h	-	>100.00h	-
		203.00	46.18K	23.44m	13.27G	42.08m	**11.55G**	>100.00h	-	18.67m	12.28G	**18.62m**	12.28G	80.93h	165.80G

Table 5.4: Comparison of time taken by different algorithms in seconds(s)/ minutes(m)/hours(h) and memory required in kilobytes(K)/ megabytes(M)/ gigabytes(G) for maintaining incremental DFS on real undirected graphs.

Dataset	n	$m \vert m^*$	$\frac{m}{n} \vert \frac{m}{m^*}$	FDFS		SDFS3		SDFS2		SDFS	
dncCoR	1.89K	5.52K	2.92	0.34s	9.22M	**0.22s**	8.61M	0.50s	**8.50M**	2.17s	**8.50M**
		4.01K	1.38	0.34s	9.22M	**0.22s**	8.62M	0.44s	8.48M	1.58s	**8.47M**
ucIrv	1.90K	20.30K	10.69	0.88s	17.47M	**0.52s**	15.30M	1.17s	22.94M	11.34s	**15.17M**
		20.12K	1.01	0.87s	17.47M	**0.49s**	15.28M	1.15s	15.16M	10.85s	**15.14M**
chess	7.30K	60.05K	8.22	26.03s	44.58M	**13.39s**	38.42M	34.00s	**38.20M**	4.46m	45.98M
		100.00	600.46	26.54s	52.48M	13.33s	**43.75M**	**0.51s**	43.94M	**0.51s**	44.64M
diggNw	30.40K	85.25K	2.80	**2.23m**	85.05M	2.98m	82.64M	8.03m	82.36M	32.33m	**81.58M**
		81.77K	1.04	**2.32m**	85.95M	3.21m	**77.41M**	8.77m	80.56M	27.70m	77.92M
asCaida	31.30K	97.84K	3.13	**35.21s**	97.45M	2.53m	87.00M	7.98m	87.95M	37.98m	**86.48M**
		122.00	801.98	35.19s	91.64M	1.99m	86.92M	2.82s	**80.75M**	**2.80s**	**80.75M**
wikiEl	7.12K	103.62K	14.55	16.21s	65.69M	**16.02s**	61.53M	41.27s	66.11M	13.83m	**56.23M**
		97.98K	1.06	16.27s	69.36M	**16.24s**	63.88M	41.16s	59.02M	14.18m	**58.25M**
slashDt	51.08K	130.37K	2.55	14.30m	127.23M	**13.85m**	123.47M	38.50m	**116.55M**	1.35h	122.88M
		84.33K	1.55	15.24m	126.61M	**14.61m**	122.08M	30.21m	**116.94M**	55.39m	122.12M
lnKMsg	27.93K	237.13K	8.49	9.52m	161.89M	**5.22m**	139.08M	12.51m	139.38M	2.02h	**138.66M**
		217.99K	1.09	9.51m	161.91M	**5.38m**	138.72M	12.35m	139.58M	2.07h	**138.66M**
fbWall	46.95K	264.00K	5.62	27.11m	200.05M	**21.08m**	175.05M	52.52m	**174.80M**	5.21h	174.81M
		263.12K	1.00	29.63m	200.03M	**22.68m**	175.03M	1.03h	**174.77M**	6.47h	174.80M
enron	87.27K	320.15K	3.67	**10.32m**	258.92M	16.00m	240.08M	58.40m	**235.11M**	11.63h	235.12M
		73.87K	4.33	**11.31m**	258.94M	16.80m	240.08M	29.48m	**234.94M**	2.64h	**234.94M**
gnutella	62.59K	501.75K	8.02	29.11m	345.39M	**23.64m**	286.66M	1.00h	**284.78M**	7.54h	284.88M
		9.00	55.75K	13.49m	288.19M	11.96m	**245.27M**	0.71s	**245.27M**	**0.69s**	245.28M
epinion	131.83K	840.80K	6.38	3.28h	556.69M	**2.50h**	487.06M	5.71h	**478.86M**	44.34h	479.22M
		939.00	895.42	3.09h	640.75M	3.04h	580.38M	**1.95m**	570.61M	1.95m	**570.59M**
digg	279.63K	1.73M	6.19	**3.42h**	1.13G	4.02h	986.58M	13.53h	**977.58M**	>100.00h	-
		1.64M	1.05	**3.23h**	1.13G	4.32h	986.58M	13.20h	**977.55M**	>100.00h	-
perLoan	89.27K	3.33M	37.31	**9.39m**	1.33G	1.11h	1.31G	4.80h	**1.31G**	>100.00h	-
		1.26K	2.65K	9.18m	1.33G	56.78m	1.31G	**4.31m**	**1.31G**	4.35m	1.31G
flickr	2.30M	33.14M	14.39	>100.00h	-	>100.00h	-	>100.00h	-	>100.00h	-
		134.00	247.31K	>100.00h	-	>100.00h	-	12.69m	**15.00G**	**12.59m**	15.00G
wikiHy	1.87M	39.95M	21.36	>100.00h	-	>100.00h	-	>100.00h	-	>100.00h	-
		2.20K	18.18K	>100.00h	-	>100.00h	-	**1.44h**	**16.99G**	1.63h	16.99G

Table 5.5: Comparison of performance of different algorithms in terms of time in seconds(s)/ minutes(m)/ hours(h) and memory required in kilobytes(K)/ megabytes(M)/ gigabytes(G) for maintaining incremental DFS on real directed graphs.

Chapter 6

Conclusion

Prior to our work, the only known algorithms for maintaining dynamic DFS were for directed acyclic graphs in the partially dynamic settings. In this thesis, we successfully addressed the problem of maintaining dynamic DFS for undirected graphs from several directions. Our results range from theoretical to experimental, sequential to parallel/ distributed/ streaming, and results having near optimal amortized to significant worst case guarantees. As a result, we now have a much better understanding about the problem in the undirected graphs. However, the problem of dynamic DFS is still wide open in various directions described as follows.

Open Questions for Dynamic DFS

Despite significant improvement of our understanding of dynamic DFS in undirected graphs, still nothing significant is known about dynamic DFS in general directed graphs. Even for any partially dynamic setting, no algorithm is known to achieve even $o(m)$ amortized bound per update, i.e., no better than recomputing the DFS tree from scratch after every update. Hence, the problem of maintaining dynamic DFS in general directed graphs is of prime significance with several applications to other dynamic graph problems. If dynamic DFS is inherently difficult for directed graphs, then even establishing non-trivial lower bounds for the same would be interesting.

The second crucial problem is to achieve better results for sparse graphs. Given the lower bound of $\Omega(n)$ per update for dynamic DFS (see Chapter 3), the existing results for dynamic DFS are no better than recomputing the DFS tree from scratch after every update when $m = O(n)$. A popular class of such sparse graphs are planar graphs, which have several interesting properties. Thus, breaking the barrier of $O(n)$ for maintaining a DFS tree for planar graphs or other special graphs that are sparse in any dynamic setting is another interesting problem worth exploring.

Open Questions on Dynamic DFS in other models

Our algorithm for maintaining dynamic DFS in distributed setting (see Chapter 4) uses a restricted $\mathcal{CONGEST}(n/D)$ model of distributed computing with large message size and greater number of messages passed. Extending this solution to more standard $\mathcal{CONGEST}$ or $\mathcal{LOCAL}$ models would truly solve the problem in the distributed setting. Moreover, even our parallel algorithms lack optimality of the total work done. Despite being time

optimal (up to $\tilde{O}(1)$ factors), our fully dynamic parallel algorithm performs $\tilde{O}(m)$ work per update which is equivalent to recomputing it from scratch. Hence, any non-trivial parallel algorithm with better bounds for work done per update would be interesting.

Finally, several graph problems have been significantly studied in these models in the static setting, whose dynamic versions are only studied in the sequential model. Thus, efficient dynamic algorithms for such problems in other models of computation would also be interesting having significant impact in practical applications.

Open questions on empirical analysis of dynamic DFS

Our experimental evaluation of incremental DFS algorithms had both empirical as well as matching theoretical results when dealing with random graphs. However, for real graphs our evaluation was restricted to the empirical analysis. Several properties of real graphs have been found to closely relate to the properties of Power Law Graphs [AB02]. Recently, some interesting theoretical studies on power law graphs [BCLS16] have shown close resemblance to the performance of algorithms on real graphs. Hence, a theoretical study of incremental DFS algorithms on power law graphs would give us a better understanding of the inferences derived from our experimental analysis on real world graphs.

Our experimental work (Chapter 5) have also opened up several promising problems which require an introspection from the empirical perspective. Subgraph connectivity is one such widely studied problem [Cha06, Dua10, CPR11], which can also be directly solved using our fully dynamic DFS algorithm (Chapter 3). Despite its practical significance, the existing results for the problem are substantially complex which may not have good empirical performance.

Appendix A

General Notations

Unless mentioned otherwise, we consider an undirected graph $G = (V, E)$ on $n = |V|$ vertices and $m = |E|$ edges. The following notations are commonly used in the thesis report.

- T : A DFS tree of G at any time during the algorithm.
- $par(v)$: Parent of v in the updated DFS tree.
- $path(u, v)$: Path between u and v in T.
- $T(x)$: The subtree of T rooted at a vertex x.
- $root(T')$: Root of a subtree T' of T, i.e., $root\big(T(x)\big) = x$.
- $LCA(u, v)$: The Lowest Common Ancestor of u and v in tree T.
- T^* : The DFS tree computed by our algorithm for the updated graph.
- $\tilde{O}()$: Hides the poly-logarithmic factors, i.e., $O(Xpoly\log n) = \tilde{O}(X)$.

A subtree T' is said to be *hanging* from a path p if the root r' of T' is a child of some vertex on the path p and r' does not belong to the path p. Unless stated otherwise, every reference to a path refers to an ancestor-descendant path defined as follows:

Definition A.1 (Ancestor-descendant path)**.** *A path p in a DFS tree T is said to be ancestor-descendant path if its endpoints have ancestor-descendant relationship in T.*

If the graph is not connected, we need to maintain a DFS tree for each connected component. However, our algorithm, at each stage, maintains a single DFS tree which stores the entire forest of these DFS trees as follows. We add a dummy vertex s (pseudo root) to the graph in the beginning and connect it to all the vertices. We maintain a DFS tree of this augmented graph rooted at s. It can be easily seen that the subtrees rooted at the children of s correspond to DFS trees of various connected components of the original graph.

Bibliography

[AA88] Alok Aggarwal and Richard J. Anderson. A random NC algorithm for depth first search. *Combinatorica*, 8(1):1–12, 1988.

[AAG87] Yehuda Afek, Baruch Awerbuch, and Eli Gafni. Applying static network protocols to dynamic networks. In *Proceedings of the 28th Annual Symposium on Foundations of Computer Science*, SFCS '87, pages 358–370, 1987.

[AAK90] Alok Aggarwal, Richard J. Anderson, and Ming-Yang Kao. Parallel depth-first search in general directed graphs. *SIAM J. Comput.*, 19(2):397–409, 1990.

[AB02] Réka Albert and Albert-László Barabási. Statistical mechanics of complex networks. *Rev. Mod. Phys.*, 74:47–97, Jan 2002.

[Abd00] Sad Abdeddaim. Algorithms and experiments on transitive closure, path cover and multiple sequence alignment. In *Algorithm Engineering and Experimentation, International Workshop ALENEX '00, San Francisco, CA, USA*, page 157169, 2000.

[ACI97a] David Alberts, Giuseppe Cattaneo, and Giuseppe F. Italiano. An empirical study of dynamic graph algorithms. *ACM Journal of Experimental Algorithmics*, 2:5, 1997.

[ACI97b] Giuseppe Amato, Giuseppe Cattaneo, and Giuseppe F. Italiano. Experimental analysis of dynamic minimum spanning tree algorithms. In *Proceedings of the Eighth Annual ACM-SIAM Symposium on Discrete Algorithms*, SODA '97, pages 314–323, 1997.

[ACK08] Baruch Awerbuch, Israel Cidon, and Shay Kutten. Optimal maintenance of a spanning tree. *J. ACM*, 55(4), 2008.

[ACK17] Ittai Abraham, Shiri Chechik, and Sebastian Krinninger. Fully dynamic all-pairs shortest paths with worst-case update-time revisited. In *Proceedings of the Twenty-Eighth Annual ACM-SIAM Symposium on Discrete Algorithms, SODA 2017, Barcelona, Spain, Hotel Porta Fira, January 16-19*, pages 440–452, 2017.

[AGM12a] Kook Jin Ahn, Sudipto Guha, and Andrew McGregor. Analyzing graph structure via linear measurements. In *Proceedings of the Twenty-Third Annual ACM-SIAM Symposium on Discrete Algorithms, SODA 2012, Kyoto, Japan, January 17-19, 2012*, pages 459–467, 2012.

[AGM12b] Kook Jin Ahn, Sudipto Guha, and Andrew McGregor. Graph sketches: sparsification, spanners, and subgraphs. In *Proceedings of the 31st ACM SIGMOD-SIGACT-SIGART Symposium on Principles of Database Systems, PODS 2012, Scottsdale, AZ, USA, May 20-24, 2012*, pages 5–14, 2012.

[AGM13] Kook Jin Ahn, Sudipto Guha, and Andrew McGregor. Spectral sparsification in dynamic graph streams. In *Approximation, Randomization, and Combinatorial Optimization. Algorithms and Techniques - 16th International Workshop, APPROX 2013, and 17th International Workshop, RANDOM 2013, Berkeley, CA, USA, August 21-23, 2013. Proceedings*, pages 1–10, 2013.

[AH00] Stephen Alstrup and Jacob Holm. Improved algorithms for finding level ancestors in dynamic trees. In *ICALP*, pages 73–84, 2000.

[AIMN91] Giorgio Ausiello, Giuseppe F. Italiano, Alberto Marchetti-Spaccamela, and Umberto Nanni. Incremental algorithms for minimal length paths. *J. Algorithms*, 12(4):615–638, 1991.

[AKL17] Sepehr Assadi, Sanjeev Khanna, and Yang Li. On estimating maximum matching size in graph streams. In *Proceedings of the Twenty-Eighth Annual ACM-SIAM Symposium on Discrete Algorithms, SODA 2017, Barcelona, Spain, Hotel Porta Fira, January 16-19*, pages 1723–1742, 2017.

[AKLY15] Sepehr Assadi, Sanjeev Khanna, Yang Li, and Grigory Yaroslavtsev. Tight bounds for linear sketches of approximate matchings. *arXiv preprint arXiv:1505.01467*, 2015.

[AKLY16] Sepehr Assadi, Sanjeev Khanna, Yang Li, and Grigory Yaroslavtsev. Maximum matchings in dynamic graph streams and the simultaneous communication model. In *ACM-SIAM Symposium on Discrete Algorithms, SODA*, pages 1345–1364, 2016.

[ALM96] Paola Alimonti, Stefano Leonardi, and Alberto Marchetti-Spaccamela. Average case analysis of fully dynamic reachability for directed graphs. *ITA*, 30(4):305–318, 1996.

[AMS99] 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.

[AS88] Baruch Awerbuch and Michael Sipser. Dynamic networks are as fast as static networks (preliminary version). In *29th Annual Symposium on Foundations of Computer Science, White Plains, New York, USA, 24-26 October 1988*, pages 206–220, 1988.

[AW14] Amir Abboud and Virginia Vassilevska Williams. Popular conjectures imply strong lower bounds for dynamic problems. In *FOCS*, pages 434–443, 2014.

[BC15] Surender Baswana and Keerti Choudhary. On dynamic DFS tree in directed graphs. In *Mathematical Foundations of Computer Science, MFCS*, pages 102–114, 2015.

[BC17a] Suman K. Bera and Amit Chakrabarti. Towards tighter space bounds for counting triangles and other substructures in graph streams. In *34th Symposium on Theoretical Aspects of Computer Science, STACS 2017, March 8-11, 2017, Hannover, Germany*, pages 11:1–11:14, 2017.

[BC17b] Aaron Bernstein and Shiri Chechik. Deterministic partially dynamic single source shortest paths for sparse graphs. In *Proceedings of the Twenty-Eighth Annual ACM-SIAM Symposium on Discrete Algorithms, SODA 2017, Barcelona, Spain, Hotel Porta Fira, January 16-19*, pages 453–469, 2017.

[BCLS16] Pawel Brach, Marek Cygan, Jakub Lacki, and Piotr Sankowski. Algorithmic complexity of power law networks. In *Proceedings of the Twenty-Seventh Annual ACM-SIAM Symposium on Discrete Algorithms, SODA 2016, Arlington, VA, USA, January 10-12, 2016*, pages 1306–1325, 2016.

[BCR16] Surender Baswana, Keerti Choudhary, and Liam Roditty. Fault tolerant subgraph for single source reachability: generic and optimal. In *Proceedings of the 48th Annual ACM SIGACT Symposium on Theory of Computing, STOC 2016, Cambridge, MA, USA, June 18-21, 2016*, pages 509–518, 2016.

[BCR17] Surender Baswana, Keerti Choudhary, and Liam Roditty. An Efficient Strongly Connected Components Algorithm in the Fault Tolerant Model. In *44th International Colloquium on Automata, Languages, and Programming (ICALP 2017)*, volume 80 of *Leibniz International Proceedings in Informatics (LIPIcs)*, pages 72:1–72:15, 2017.

[Ber09] Aaron Bernstein. Fully dynamic (2 + epsilon) approximate all-pairs shortest paths with fast query and close to linear update time. In *50th Annual IEEE Symposium on Foundations of Computer Science, FOCS 2009, October 25-27, 2009, Atlanta, Georgia, USA*, pages 693–702, 2009.

[Ber16] Aaron Bernstein. Maintaining shortest paths under deletions in weighted directed graphs. *SIAM J. Comput.*, 45(2):548–574, 2016.

[Ber17] Aaron Bernstein. Deterministic partially dynamic single source shortest paths in weighted graphs. In *44th International Colloquium on Automata, Languages, and Programming, ICALP 2017, July 10-14, 2017, Warsaw, Poland*, pages 44:1–44:14, 2017.

[BGPW17] Greg Bodwin, Fabrizio Grandoni, Merav Parter, and Virginia Vassilevska Williams. Preserving distances in very faulty graphs. In *44th International Colloquium on Automata, Languages, and Programming, ICALP 2017, July 10-14, 2017, Warsaw, Poland*, pages 73:1–73:14, 2017.

[BHNT15] Sayan Bhattacharya, Monika Henzinger, Danupon Nanongkai, and Charalampos Tsourakakis. Space-and time-efficient algorithm for maintaining dense subgraphs on one-pass dynamic streams. In *Proceedings of the forty-seventh annual ACM symposium on Theory of computing*, pages 173–182. ACM, 2015.

[BK09] Aaron Bernstein and David R. Karger. A nearly optimal oracle for avoiding failed vertices and edges. In *Proceedings of the 41st Annual ACM Symposium on Theory of Computing, STOC 2009, Bethesda, MD, USA, May 31 - June 2, 2009*, pages 101–110, 2009.

[BKLvR09] Ulrik Brandes, Patrick Kenis, Jrgen Lerner, and Denise van Raaij. Computing wikipedia edit networks. *Technical Report*, 2009.

[BKS12] Surender Baswana, Sumeet Khurana, and Soumojit Sarkar. Fully dynamic randomized algorithms for graph spanners. *ACM Transactions on Algorithms*, 8(4):35, 2012.

[BMM14] Glencora Borradaile, Claire Mathieu, and Theresa Migler. Lower bounds for testing digraph connectivity with one-pass streaming algorithms. *CoRR*, abs/1404.1323, 2014.

[BMST06] H. Bast, Kurt Mehlhorn, Guido Schäfer, and Hisao Tamaki. Matching algorithms are fast in sparse random graphs. *Theory Comput. Syst.*, 39(1):3–14, 2006.

[Bol84] Béla Bollobs. The evolution of random graphs. *Transactions of the American Mathematical Society*, 286 (1):257–274, 1984.

[BR11] Aaron Bernstein and Liam Roditty. Improved dynamic algorithms for maintaining approximate shortest paths under deletions. In *SODA*, pages 1355–1365, 2011.

[BRT08] Luciana S. Buriol, Mauricio G. C. Resende, and Mikkel Thorup. Speeding up dynamic shortest-path algorithms. *INFORMS J. on Computing*, 20(2):191–204, April 2008.

[BS15] Marc Bury and Chris Schwiegelshohn. Sublinear estimation of weighted matchings in dynamic data streams. In *Algorithms - ESA 2015 - 23rd Annual European Symposium, Patras, Greece, September 14-16, 2015, Proceedings*, pages 263–274, 2015.

[BW09] Reinhard Bauer and Dorothea Wagner. Batch dynamic single-source shortest-path algorithms: An experimental study. In *Experimental Algorithms, 8th International Symposium, SEA 2009, Dortmund, Germany, June 4-6, 2009. Proceedings*, pages 51–62, 2009.

[CCE+16] Rajesh Chitnis, Graham Cormode, Hossein Esfandiari, MohammadTaghi Hajiaghayi, Andrew McGregor, Morteza Monemizadeh, and Sofya Vorotnikova. Kernelization via sampling with applications to finding matchings and related problems in dynamic graph streams. In *Proceedings of the Twenty-Seventh Annual ACM-SIAM Symposium on Discrete Algorithms, SODA 2016, Arlington, VA, USA, January 10-12, 2016*, pages 1326–1344, 2016.

[CDSF10] Serafino Cicerone, Gianlorenzo D'Angelo, Gabriele Di Stefano, and Daniele Frigioni. Partially dynamic efficient algorithms for distributed shortest paths. *Theor. Comput. Sci.*, 411(7-9):1013–1037, 2010.

[CFPI10] Giuseppe Cattaneo, Pompeo Faruolo, Umberto Ferraro Petrillo, and Giuseppe F. Italiano. Maintaining dynamic minimum spanning trees: An experimental study. *Discrete Applied Mathematics*, 158(5):404–425, 2010.

[CH05] Richard Cole and Ramesh Hariharan. Dynamic lca queries on trees. *SIAM J. Comput.*, 34(4):894–923, 2005.

[Cha90] P. Chaudhuri. Finding and updating depth-first spanning trees of acyclic digraphs in parallel. *The Computer Journal*, 33(3):247–251, 1990.

[Cha06] Timothy M. Chan. Dynamic subgraph connectivity with geometric applications. *SIAM J. Comput.*, 36(3):681–694, 2006.

[CHHK16] Keren Censor-Hillel, Elad Haramaty, and Zohar Karnin. Optimal dynamic distributed mis. In *Proceedings of the 2016 ACM Symposium on Principles of Distributed Computing*, PODC '16, 2016.

[CHI+16] Shiri Chechik, Thomas Dueholm Hansen, Giuseppe F. Italiano, Jakub Lacki, and Nikos Parotsidis. Decremental single-source reachability and strongly connected components in $\tilde{o}(m\sqrt{n})$ total update time. In *IEEE 57th Annual Symposium on Foundations of Computer Science, FOCS 2016, 9-11 October 2016, Hyatt Regency, New Brunswick, New Jersey, USA*, pages 315–324, 2016.

[Cid88] Israel Cidon. Yet another distributed depth-first-search algorithm. *Inf. Process. Lett.*, 26(6):301–305, 1988.

[CLRS09] Thomas H. Cormen, Charles E. Leiserson, Ronald L. Rivest, and Clifford Stein. *Introduction to Algorithms (3. ed.)*. MIT Press, 2009.

[CMP+10] Daniel Cordeiro, Grégory Mounié, Swann Perarnau, Denis Trystram, Jean-Marc Vincent, and Frédéric Wagner. Random graph generation for scheduling simulations. In *3rd International Conference on Simulation Tools and Techniques, SIMUTools '10, Malaga, Spain - March 16 - 18, 2010*, page 60, 2010.

[CMS13] Michael S. Crouch, Andrew McGregor, and Daniel Stubbs. Dynamic graphs in the sliding-window model. In *Algorithms - ESA 2013 - 21st Annual European Symposium, Sophia Antipolis, France, September 2-4, 2013. Proceedings*, pages 337–348, 2013.

[Col88] Richard Cole. Parallel merge sort. *SIAM J. Comput.*, 17(4):770–785, 1988.

[Col93] Richard Cole. Correction: Parallel merge sort. *SIAM J. Comput.*, 22(6):1349, 1993.

[CPR08] Timothy M. Chan, Mihai Patrascu, and Liam Roditty. Dynamic connectivity: Connecting to networks and geometry. In *49th Annual IEEE Symposium on Foundations of Computer Science, FOCS 2008*, pages 95–104, 2008.

[CPR11] Timothy M. Chan, Mihai Patrascu, and Liam Roditty. Dynamic connectivity: Connecting to networks and geometry. *SIAM J. Comput.*, 40(2):333–349, 2011.

[Cro91] Steven T. Crocker. An experimental comparison of two maximum cardinality matching programs. In *Network Flows And Matching, Proceedings of a DIMACS Workshop, New Brunswick, New Jersey, USA, October 14-16, 1991*, pages 519–538, 1991.

[CSFN03] Serafino Cicerone, Gabriele Di Stefano, Daniele Frigioni, and Umberto Nanni. A fully dynamic algorithm for distributed shortest paths. *Theor. Comput. Sci.*, 297(1-3):83–102, 2003.

[CSJS09] Munmun De Choudhury, Hari Sundaram, Ajita John, and Dorée Duncan Seligmann. Social synchrony: Predicting mimicry of user actions in online social media. In *Proc. Int. Conf. on Computational Science and Engineering*, pages 151–158, 2009.

[DDF+15] Annalisa D'Andrea, Mattia D'Emidio, Daniele Frigioni, Stefano Leucci, and Guido Proietti. Dynamic maintenance of a shortest-path tree on homogeneous batches of updates: New algorithms and experiments. *ACM Journal of Experimental Algorithmics*, 20:1.5:1.1–1.5:1.33, 2015.

[DF91] Martin E. Dyer and Alan M. Frieze. Probabilistic analysis of a parallel algorithm for finding the lexicographically first depth first search tree in a dense random graph. *Random Struct. Algorithms*, 2(2):233–240, 1991.

[DF99] Sajal K. Das and Paolo Ferragina. An EREW PRAM algorithm for updating minimum spanning trees. *Parallel Processing Letters*, 9(1):111–122, 1999.

[DFIT06] Camil Demetrescu, Pompeo Faruolo, Giuseppe F. Italiano, and Mikkel Thorup. Does path cleaning help in dynamic all-pairs shortest paths? In *Algorithms - ESA 2006, 14th Annual European Symposium, Zurich, Switzerland, September 11-13, 2006, Proceedings*, pages 732–743, 2006.

[DFMN00] Camil Demetrescu, Daniele Frigioni, Alberto Marchetti-Spaccamela, and Umberto Nanni. Maintaining shortest paths in digraphs with arbitrary arc weights: An experimental study. In *Algorithm Engineering, 4th International Workshop, WAE 2000, Saarbrücken, Germany, September 5-8, 2000, Proceedings*, pages 218–229, 2000.

[DH14] Erik Demaine and Mohammad Taghi Hajiaghayi. Bigdnd: Big dynamic network data. `http://projects.csail.mit.edu/dnd/`, 2014.

[DI04] Camil Demetrescu and Giuseppe F. Italiano. A new approach to dynamic all pairs shortest paths. *J. ACM*, 51(6):968–992, 2004.

[DI05] Camil Demetrescu and Giuseppe F. Italiano. Trade-offs for fully dynamic transitive closure on dags: breaking through the o(n^2 barrier. *J. ACM*, 52(2):147–156, 2005.

[DI06a] Camil Demetrescu and Giuseppe F. Italiano. Dynamic shortest paths and transitive closure: Algorithmic techniques and data structures. *J. Discrete Algorithms*, 4(3):353–383, 2006.

[DI06b] Camil Demetrescu and Giuseppe F. Italiano. Experimental analysis of dynamic all pairs shortest path algorithms. *ACM Trans. Algorithms*, 2(4):578–601, 2006.

[DI08] Camil Demetrescu and Giuseppe F. Italiano. Mantaining dynamic matrices for fully dynamic transitive closure. *Algorithmica*, 51(4):387–427, 2008.

[DLPP13] Ajoy Kumar Datta, Lawrence L. Larmore, Linda Pagli, and Giuseppe Prencipe. Linear time distributed swap edge algorithms. In *Algorithms and Complexity, 8th International Conference, CIAC 2013, Barcelona, Spain, May 22-24, 2013. Proceedings*, pages 122–133, 2013.

[DP09] Ran Duan and Seth Pettie. Dual-failure distance and connectivity oracles. In *Proceedings of the Twentieth Annual ACM-SIAM Symposium on Discrete Algorithms, SODA 2009, New York, NY, USA, January 4-6, 2009*, pages 506–515, 2009.

[DP10] Ran Duan and Seth Pettie. Connectivity oracles for failure prone graphs. In *Proceedings of the 42nd ACM Symposium on Theory of Computing, STOC 2010*, pages 465–474, 2010.

[DP17] Ran Duan and Seth Pettie. Connectivity oracles for graphs subject to vertex failures. In *Proceedings of the Twenty-Eighth Annual ACM-SIAM Symposium on Discrete Algorithms, SODA 2017, Barcelona, Spain, Hotel Porta Fira, January 16-19*, pages 490–509, 2017.

[DTCR08] Camil Demetrescu, Mikkel Thorup, Rezaul Alam Chowdhury, and Vijaya Ramachandran. Oracles for distances avoiding a failed node or link. *SIAM J. Comput.*, 37(5):1299–1318, 2008.

[Dua10] Ran Duan. New data structures for subgraph connectivity. In *Automata, Languages and Programming, 37th International Colloquium, ICALP 2010, Proceedings, Part I*, pages 201–212, 2010.

[DW07] Daniel Delling and Dorothea Wagner. Landmark-based routing in dynamic graphs. In *Proceedings of the 6th International Conference on Experimental Algorithms*, WEA'07, pages 52–65, 2007.

[DZ17] Ran Duan and Le Zhang. Faster randomized worst-case update time for dynamic subgraph connectivity. In *Algorithms and Data Structures - 15th International Symposium, WADS 2017, St. John's, NL, Canada, July 31 - August 2, 2017, Proceedings*, pages 337–348, 2017.

[EGIN97] David Eppstein, Zvi Galil, Giuseppe F. Italiano, and Amnon Nissenzweig. Sparsification - a technique for speeding up dynamic graph algorithms. *J. ACM*, 44(5):669–696, 1997.

[EHW16] Hossein Esfandiari, MohammadTaghi Hajiaghayi, and David P. Woodruff. Brief announcement: Applications of uniform sampling: Densest subgraph and beyond. In *Proceedings of the 28th ACM Symposium on Parallelism in Algorithms and Architectures, SPAA 2016, Asilomar State Beach/Pacific Grove, CA, USA, July 11-13, 2016*, pages 397–399, 2016.

[Elk07] Michael Elkin. A near-optimal distributed fully dynamic algorithm for maintaining sparse spanners. In *Proceedings of the Twenty-sixth Annual ACM Symposium on Principles of Distributed Computing*, PODC '07, pages 185–194, 2007.

[ELS15] Alessandro Epasto, Silvio Lattanzi, and Mauro Sozio. Efficient densest subgraph computation in evolving graphs. In *Proceedings of the 24th International Conference on World Wide Web, WWW 2015, Florence, Italy, May 18-22, 2015*, pages 300–310, 2015.

[ER59] Paul Erdős and Alfréd Rényi. On random graphs I. *Publicationes Mathematicae (Debrecen)*, 6:290–297, 1959.

[ER60] P. Erdős and A Rényi. On the evolution of random graphs. In *Publication of the Mathematical Institute of the Hungarian Academy of Sciences*, pages 17–61, 1960.

[ES81] Shimon Even and Yossi Shiloach. An on-line edge-deletion problem. *J. ACM*, 28(1):1–4, 1981.

[ET75] Shimon Even and Robert Endre Tarjan. Network flow and testing graph connectivity. *SIAM J. Comput.*, 4(4):507–518, 1975.

[FEP+12] Paola Flocchini, Toni Mesa Enriquez, Linda Pagli, Giuseppe Prencipe, and Nicola Santoro. Distributed minimum spanning tree maintenance for transient node failures. *IEEE Trans. Computers*, 61(3):408–414, 2012.

[Fer95] P. Ferragina. A technique to speed up parallel fully dynamic algorithms for MST. *J. Parallel Distrib. Comput.*, 31(2):181–189, December 1995.

[FGN97] Paolo Giulio Franciosa, Giorgio Gambosi, and Umberto Nanni. The incremental maintenance of a depth-first-search tree in directed acyclic graphs. *Inf. Process. Lett.*, 61(2):113–120, 1997.

[FHLT15] 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, 2015.

[FINP98] Daniele Frigioni, Mario Ioffreda, Umberto Nanni, and Giulio Pasquale. Experimental analysis of dynamic algorithms for the single-source shortest-path problem. *ACM Journal of Experimental Algorithmics*, 3:5, 1998.

[FK15] A. Frieze and M. Karoński. *Introduction to Random Graphs.* Cambridge University Press, 2015.

[FKM+05] 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.

[FKSV03] 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.

[FL96] Paolo Ferragina and Fabrizio Luccio. Three techniques for parallel maintenance of a minimum spanning tree under batch of updates. *Parallel Processing Letters*, 6(2):213–222, 1996.

[FMNZ01] Daniele Frigioni, Tobias Miller, Umberto Nanni, and Christos D. Zaroliagis. An experimental study of dynamic algorithms for transitive closure. *ACM Journal of Experimental Algorithmics*, 6:9, 2001.

[Fre85] Greg N. Frederickson. Data structures for on-line updating of minimum spanning trees, with applications. *SIAM J. Comput.*, 14(4):781–798, 1985.

[Fre91] Jon Freeman. Parallel algorithms for depth-first search. *Technical Report, University of Pennsylvania*, October 1991.

[FSZZ99] Panagiota Fatourou, Paul G. Spirakis, Panagiotis Zarafidis, and Anna Zoura. Implementation an experimental evaluation of graph connectivity algorithms using LEDA. In *Algorithm Engineering, 3rd International Workshop, WAE '99, London, UK, July 19-21, 1999, Proceedings*, pages 124–138, 1999.

[FW78] Steven Fortune and James Wyllie. Parallelism in random access machines. In *Proceedings of the Tenth Annual ACM Symposium on Theory of Computing*, STOC '78, pages 114–118, 1978.

[Gaf87] Eli Gafni. Generalized scheme for topology-update in dynamic networks. In *Distributed Algorithms, 2nd International Workshop, Amsterdam, The Netherlands, July 8-10, 1987, Proceedings*, pages 187–196, 1987.

[GB84] Ratan K. Ghosh and G. P. Bhattacharjee. A parallel search algorithm for directed acyclic graphs. *BIT*, 24(2):134–150, 1984.

[GHK+15] Andrew V. Goldberg, Sagi Hed, Haim Kaplan, Pushmeet Kohli, Robert Endre Tarjan, and Renato F. Werneck. Faster and more dynamic maximum flow by incremental breadth-first search. In *Algorithms - ESA 2015 - 23rd Annual European Symposium, Patras, Greece, September 14-16, 2015, Proceedings*, pages 619–630, 2015.

[GILS12] Loukas Georgiadis, Giuseppe F. Italiano, Luigi Laura, and Federico Santaroni. An experimental study of dynamic dominators. In *Algorithms - ESA 2012 - 20th Annual European Symposium, Ljubljana, Slovenia, September 10-12, 2012. Proceedings*, pages 491–502, 2012.

[GK17] Manoj Gupta and Shahbaz Khan. Multiple source dual fault tolerant BFS trees. In *44th International Colloquium on Automata, Languages, and Programming, ICALP 2017, July 10-14, 2017, Warsaw, Poland*, pages 127:1–127:15, 2017.

[GKL08] Vicen Gómez, Andreas Kaltenbrunner, and Vicente López. Statistical analysis of the social network and discussion threads in Slashdot. In *Proc. Int. World Wide Web Conf.*, pages 645–654, 2008.

[GKP12] Ashish Goel, Michael Kapralov, and Ian Post. Single pass sparsification in the streaming model with edge deletions. *CoRR*, abs/1203.4900, 2012.

[GKS01] 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.

[GMT15] Sudipto Guha, Andrew McGregor, and David Tench. Vertex and hyperedge connectivity in dynamic graph streams. In *ACM Symposium on Principles of Database Systems, PODS*, pages 241–247, 2015.

[GP16] Mohsen Ghaffari and Merav Parter. Near-optimal distributed algorithms for fault-tolerant tree structures. In *Proceedings of the 28th ACM Symposium on Parallelism in Algorithms and Architectures, SPAA 2016, Asilomar State Beach/Pacific Grove, CA, USA, July 11-13, 2016*, pages 387–396, 2016.

[GPV93] Andrew V. Goldberg, Serge A. Plotkin, and Pravin M. Vaidya. Sublinear-time parallel algorithms for matching and related problems. *J. Algorithms*, 14(2):180–213, 1993.

[GSW11] Beat Gfeller, Nicola Santoro, and Peter Widmayer. A distributed algorithm for finding all best swap edges of a minimum-diameter spanning tree. *IEEE Trans. Dependable Sec. Comput.*, 8(1):1–12, 2011.

[GW12] Fabrizio Grandoni and Virginia Vassilevska Williams. Improved distance sensitivity oracles via fast single-source replacement paths. In *53rd Annual IEEE Symposium on Foundations of Computer Science, FOCS 2012, New Brunswick, NJ, USA, October 20-23, 2012*, pages 748–757, 2012.

[Hag90] Torben Hagerup. Planar depth-first search in O(log n) parallel time. *SIAM J. Comput.*, 19(4):678–704, 1990.

[HdLT01] Jacob Holm, Kristian de Lichtenberg, and Mikkel Thorup. Poly-logarithmic deterministic fully-dynamic algorithms for connectivity, minimum spanning tree, 2-edge, and biconnectivity. *J. ACM*, 48(4):723–760, 2001.

[Hen95] Monika Rauch Henzinger. Fully dynamic biconnectivity in graphs. *Algorithmica*, 13(6):503–538, 1995.

[Hen00] Monika Rauch Henzinger. Improved data structures for fully dynamic biconnectivity. *SIAM J. Comput.*, 29(6):1761–1815, 2000.

[HF98] Monika Rauch Henzinger and Michael L. Fredman. Lower bounds for fully dynamic connectivity problems in graphs. *Algorithmica*, 22(3):351–362, 1998.

[HHKP17] Shang-En Huang, Dawei Huang, Tsvi Kopelowitz, and Seth Pettie. Fully dynamic connectivity in $O(\log n(\log\log n)^2)$ amortized expected time. In *Proceedings of the Twenty-Eighth Annual ACM-SIAM Symposium on Discrete Algorithms, SODA 2017, Barcelona, Spain, Hotel Porta Fira, January 16-19*, pages 510–520, 2017.

[HK73] 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.

[HK95] M. R. Henzinger and V. King. Fully dynamic biconnectivity and transitive closure. In *Proceedings of the 36th Annual Symposium on Foundations of Computer Science*, FOCS '95, pages 664–, 1995.

[HK99] Monika Rauch Henzinger and Valerie King. Randomized fully dynamic graph algorithms with polylogarithmic time per operation. *J. ACM*, 46(4):502–516, 1999.

[HKN13] Monika Henzinger, Sebastian Krinninger, and Danupon Nanongkai. Sublinear-time maintenance of breadth-first spanning tree in partially dynamic networks. In *ICALP (2)*, pages 607–619, 2013.

[HKN14a] Monika Henzinger, Sebastian Krinninger, and Danupon Nanongkai. Decremental single-source shortest paths on undirected graphs in near-linear total update time. In *55th IEEE Annual Symposium on Foundations of Computer Science, FOCS 2014, Philadelphia, PA, USA, October 18-21, 2014*, pages 146–155, 2014.

[HKN14b] Monika Henzinger, Sebastian Krinninger, and Danupon Nanongkai. Sublinear-time decremental algorithms for single-source reachability and shortest paths on directed graphs. In *Symposium on Theory of Computing, STOC 2014, New York, NY, USA, May 31 - June 03, 2014*, pages 674–683, 2014.

[HKN15] Monika Henzinger, Sebastian Krinninger, and Danupon Nanongkai. Improved algorithms for decremental single-source reachability on directed graphs. In *Automata, Languages, and Programming - 42nd International Colloquium, ICALP 2015, Kyoto, Japan, July 6-10, 2015, Proceedings, Part I*, pages 725–736, 2015.

[HKN16] Monika Henzinger, Sebastian Krinninger, and Danupon Nanongkai. Dynamic approximate all-pairs shortest paths: Breaking the o(mn) barrier and derandomization. *SIAM J. Comput.*, 45(3):947–1006, 2016.

[HKNS15] Monika Henzinger, Sebastian Krinninger, Danupon Nanongkai, and Thatchaphol Saranurak. Unifying and strengthening hardness for dynamic problems via the online matrix-vector multiplication conjecture. In *STOC*, pages 21–30, 2015.

[HL12] T. Hogg and K. Lerman. Social dynamics of Digg. *EPJ Data Science*, 1(5), 2012.

[HRW15] Jacob Holm, Eva Rotenberg, and Christian Wulff-Nilsen. Faster fully-dynamic minimum spanning forest. In *Algorithms - ESA 2015 - 23rd Annual European Symposium, Patras, Greece, September 14-16, 2015, Proceedings*, pages 742–753, 2015.

[HS17] Michael Huang and Clifford Stein. Extending search phases in the Micali-Vazirani algorithm. In *16th International Symposium on Experimental Algorithms, SEA 2017, June 21-23, 2017, London, UK*, pages 10:1–10:19, 2017.

[HT74] John E. Hopcroft and Robert Endre Tarjan. Efficient planarity testing. *J. ACM*, 21(4):549–568, 1974.

[IKRT01] Raj Iyer, David R. Karger, Hariharan Rahul, and Mikkel Thorup. An experimental study of polylogarithmic, fully dynamic, connectivity algorithms. *ACM Journal of Experimental Algorithmics*, 6:4, 2001.

[Ita86] Giuseppe F. Italiano. Amortized efficiency of a path retrieval data structure. *Theor. Comput. Sci.*, 48(3):273–281, 1986.

[Ita91] Giuseppe F. Italiano. Distributed algorithms for updating shortest paths (extended abstract). In *Distributed Algorithms, 5th International Workshop, WDAG '91, Delphi, Greece, October 7-9, 1991, Proceedings*, pages 200–211, 1991.

[JáJ92] Joseph JáJá. *An Introduction to Parallel Algorithms*. Addison-Wesley, 1992.

[Jan14] Abhabongse Janthong. Streaming algorithm for determining a topological ordering of a digraph. *UG Thesis, Brown University*, 2014.

[Jen97] Esther Jennings. Distributed computation and maintenance of 3-edge-connected components during edge insertions. In *Structure, Information and Communication Complexity: 3rd Colloquium, SIROCCO'96, Certosa Di Pontignano, Siena, June 1996: Proceedings*, volume 6, page 224, 1997.

[JM88] Hermann Jung and Kurt Mehlhorn. Parallel algorithms for computing maximal independent sets in trees and for updating minimum spanning trees. *Inf. Process. Lett.*, 27(5):227–236, 1988.

[JM96] Donald B. Johnson and Panagiotis Takis Metaxas. Optimal algorithms for the single and multiple vertex updating problems of a minimum spanning tree. *Algorithmica*, 16(6):633–648, 1996.

[Kao88] Ming-Yang Kao. All graphs have cycle separators and planar directed depth-first search is in DNC. In *VLSI Algorithms and Architectures, 3rd Aegean Workshop on Computing, AWOC, Proceedings*, pages 53–63, 1988.

[Kap90] S. Kapidakis. Average-case analysis of graph-searching algorithms. *PhD Thesis, Princeton University*, no. 286, 1990.

[KC86] Taenam Kim and Kyungyong Chwa. Parallel algorithms for a depth first search and a breadth first search. *International Journal of Computer Mathematics*, 19(1):39–54, 1986.

[Kin99] Valerie King. Fully dynamic algorithms for maintaining all-pairs shortest paths and transitive closure in digraphs. In *Proceedings of the 40th Annual Symposium on Foundations of Computer Science*, FOCS '99, pages 81–, 1999.

[KIS90] Devendra Kumar, S. Sitharama Iyengar, and Mohan B. Sharma. Corrigenda: Corrections to a distributed depth-first search algorithm. *Inf. Process. Lett.*, 35(1):55–56, 1990.

[KKM13] Bruce M. Kapron, Valerie King, and Ben Mountjoy. Dynamic graph connectivity in polylogarithmic worst case time. In *SODA*, pages 1131–1141, 2013.

[KKPT16] Casper Kejlberg-Rasmussen, Tsvi Kopelowitz, Seth Pettie, and Mikkel Thorup. Faster worst case deterministic dynamic connectivity. In *24th Annual European Symposium on Algorithms, ESA 2016, August 22-24, 2016, Aarhus, Denmark*, pages 53:1–53:15, 2016.

[KKT15] Valerie King, Shay Kutten, and Mikkel Thorup. 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*, pages 71–80, 2015.

[KLM+17] Michael Kapralov, Yin Tat Lee, Cameron Musco, Christopher Musco, and Aaron Sidford. Single pass spectral sparsification in dynamic streams. *SIAM J. Comput.*, 46(1):456–477, 2017.

[Kon15] Christian Konrad. Maximum matching in turnstile streams. In *Algorithms - ESA 2015 - 23rd Annual European Symposium, Patras, Greece, September 14-16, 2015, Proceedings*, pages 840–852, 2015.

[KP98] John D. Kececioglu and A. Justin Pecqueur. Computing maximum-cardinality matchings in sparse general graphs. In *Algorithm Engineering, 2nd International Workshop, WAE '92, Saarbrücken, Germany, August 20-22, 1998, Proceedings*, pages 121–132, 1998.

[KP17] John Kallaugher and Eric Price. A hybrid sampling scheme for triangle counting. In *Proceedings of the Twenty-Eighth Annual ACM-SIAM Symposium on Discrete Algorithms, SODA 2017, Barcelona, Spain, Hotel Porta Fira, January 16-19*, pages 1778–1797, 2017.

[Kru56] Joseph B. Kruskal. On the Shortest Spanning Subtree of a Graph and the Traveling Salesman Problem. In *Proceedings of the American Mathematical Society, 7*, volume 1, pages 48–15, 1956.

[KS13] Michael Krivelevich and Benny Sudakov. The phase transition in random graphs: A simple proof. *Random Struct. Algorithms*, 43(2):131–138, 2013.

[KT01] Valerie King and Mikkel Thorup. A space saving trick for directed dynamic transitive closure and shortest path algorithms. In *Proceedings of the 7th Annual International Conference on Computing and Combinatorics*, COCOON '01, pages 268–277, 2001.

[KT07] Pushmeet Kohli and Philip H. S. Torr. Dynamic graph cuts for efficient inference in markov random fields. *IEEE Trans. Pattern Anal. Mach. Intell.*, 29(12):2079–2088, 2007.

[KT08] Pushmeet Kohli and Philip H. S. Torr. Measuring uncertainty in graph cut solutions. *Computer Vision and Image Understanding*, 112(1):30–38, 2008.

[Kun16] Jérôme Kunegis. Konect - the koblenz network collection. `http://konect.uni-koblenz.de/networks/`, October 2016.

[KW14] Michael Kapralov and David P. Woodruff. Spanners and sparsifiers in dynamic streams. In *ACM Symposium on Principles of Distributed Computing, PODC '14, Paris, France, July 15-18, 2014*, pages 272–281, 2014.

[KY04] Bryan Klimt and Yiming Yang. The Enron corpus: A new dataset for email classification research. In *Proc. European Conf. on Machine Learning*, pages 217–226, 2004.

[KZ08] Ioannis Krommidas and Christos D. Zaroliagis. An experimental study of algorithms for fully dynamic transitive closure. *ACM Journal of Experimental Algorithmics*, 12:1.6:1–1.6:22, 2008.

[Lac13] Jakub Lacki. Improved deterministic algorithms for decremental reachability and strongly connected components. *ACM Trans. Algorithms*, 9(3):27:1–27:15, 2013.

[LBS01] Weifa Liang, Richard P. Brent, and Hong Shen. Fully dynamic maintenance of k-connectivity in parallel. *IEEE Trans. Parallel Distrib. Syst.*, 12(8), August 2001.

[LEK08] Shuang Liu, Brice Effantin, and Hamamache Kheddouci. A fully dynamic distributed algorithm for a B-coloring of graphs. In *IEEE International Symposium on Parallel and Distributed Processing with Applications, ISPA*, pages 657–662, 2008.

[Ley02] Michael Ley. The DBLP computer science bibliography: Evolution, research issues, perspectives. In *Proc. Int. Symposium on String Processing and Information Retrieval*, pages 1–10, 2002.

[LHK10] Jure Leskovec, Daniel Huttenlocher, and Jon Kleinberg. Governance in social media: A case study of the Wikipedia promotion process. In *Proc. Int. Conf. on Weblogs and Social Media*, 2010.

[LK14] Jure Leskovec and Andrej Krevl. SNAP Datasets: Stanford large network dataset collection. `http://snap.stanford.edu/data/`, June 2014.

[LKF05] Jure Leskovec, Jon M. Kleinberg, and Christos Faloutsos. Graphs over time: densification laws, shrinking diameters and possible explanations. In *Proceedings of the Eleventh ACM SIGKDD International Conference on Knowledge Discovery and Data Mining, Chicago, Illinois, USA, August 21-24, 2005*, pages 177–187, 2005.

[LKF07] Jure Leskovec, Jon Kleinberg, and Christos Faloutsos. Graph evolution: Densification and shrinking diameters. *ACM Trans. Knowledge Discovery from Data*, 1(1):1–40, 2007.

[LMS96] Weifa Liang, Brendan D. McKay, and Hong Shen. NC algorithms for dynamically solving the all pairs shortest paths problem and related problems. *Inf. Process. Lett.*, 58(3):149–155, 1996.

[LMT87] K. B. Lakshmanan, N. Meenakshi, and Krishnaiyan Thulasiraman. A time-optimal message-efficient distributed algorithm for depth-first-search. *Inf. Process. Lett.*, 25(2):103–109, 1987.

[LPvL88] J. A. La Poutré and J. van Leeuwen. Maintenance of transitive closures and transitive reductions of graphs. In *Proceedings of the International Workshop WG '87 on Graph-theoretic Concepts in Computer Science*, pages 106–120, 1988.

[LS95] Weifa Liang and Hong Shen. Fully dynamic maintaining 2-edge connectivity in parallel. In *Proceedings of the 7th IEEE Symposium on Parallel and Distributeed Processing*, SPDP '95, pages 216–, 1995.

[MA05] Paolo Massa and Paolo Avesani. Controversial users demand local trust metrics: an experimental study on epinions.com community. In *Proc. American Association for Artificial Intelligence Conf.*, pages 121–126, 2005.

[Mar17] Daniel P. Martin. Dynamic Shortest Path and Transitive Closure Algorithms: A Survey. *ArXiv e-prints*, 2017.

[McG14] Andrew McGregor. Graph stream algorithms: A survey. *SIGMOD Rec.*, 43(1):9–20, May 2014.

[McG17] Andrew McGregor. Graph sketching and streaming: New approaches for analyzing massive graphs. In *Computer Science - Theory and Applications - 12th International Computer Science Symposium in Russia, CSR 2017, Kazan, Russia, June 8-12, 2017, Proceedings*, pages 20–24, 2017.

[Mel06] Guy Melancon. Just how dense are dense graphs in the real world?: A methodological note. In *Proceedings of the 2006 AVI Workshop on BEyond Time and Errors: Novel Evaluation Methods for Information Visualization*, BELIV '06, pages 1–7, 2006.

[Mey01] Ulrich Meyer. Single-source shortest-paths on arbitrary directed graphs in linear average-case time. In *Proceedings of the Twelfth Annual Symposium on Discrete Algorithms, January 7-9, 2001, Washington, DC, USA.*, pages 797–806, 2001.

[MH96] S. A. M. Makki and George Havas. Distributed algorithms for depth-first search. *Inf. Process. Lett.*, 60(1):7–12, 1996.

[Mis09] Alan Mislove. *Online Social Networks: Measurement, Analysis, and Applications to Distributed Information Systems.* PhD thesis, Rice University, Department of Computer Science, May 2009.

[MKG+08] Alan Mislove, Hema Swetha Koppula, Krishna P. Gummadi, Peter Druschel, and Bobby Bhattacharjee. Growth of the flickr social network. In *Proceedings of the 1st ACM SIGCOMM Workshop on Social Networks (WOSN'08)*, August 2008.

[Mot94] Rajeev Motwani. Average-case analysis of algorithms for matchings and related problems. *J. ACM*, 41(6):1329–1356, 1994.

[MR91] R. Bruce Mattingly and Nathan P. Ritchey. Implementing on O(/NM) cardinality matching algorithm. In *Network Flows And Matching, Proceedings of a DIMACS Workshop, New Brunswick, New Jersey, USA, October 14-16, 1991*, pages 539–556, 1991.

[MSVT94] Peter Bro Miltersen, Sairam Subramanian, Jeffrey Scott Vitter, and Roberto Tamassia. Complexity models for incremental computation. *Theor. Comput. Sci.*, 130(1):203–236, 1994.

[MTVV15] Andrew McGregor, David Tench, Sofya Vorotnikova, and Hoa T. Vu. Densest subgraph in dynamic graph streams. In *Mathematical Foundations of Computer Science 2015 - 40th International Symposium, MFCS 2015, Milan, Italy, August 24-28, 2015, Proceedings, Part II*, pages 472–482, 2015.

[Mut05] S. Muthukrishnan. Data streams: Algorithms and applications. *Foundations and Trends in Theoretical Computer Science*, 1(2):117–236, 2005.

[MV80] Silvio Micali and Vijay V. Vazirani. An o(sqrt(|v|) |e|) algorithm for finding maximum matching in general graphs. In *21st Annual Symposium on Foundations of Computer Science, Syracuse, New York, USA, 13-15 October 1980*, pages 17–27, 1980.

[MVV16] Andrew McGregor, Sofya Vorotnikova, and Hoa T. Vu. Better algorithms for counting triangles in data streams. In *Proceedings of the 35th ACM SIGMOD-SIGACT-SIGAI Symposium on Principles of Database Systems, PODS 2016, San Francisco, CA, USA, June 26 - July 01, 2016*, pages 401–411, 2016.

[NRSY95] Sotiris E. Nikoletseas, John H. Reif, Paul G. Spirakis, and Moti Yung. Stochastic graphs have short memory: Fully dynamic connectivity in poly-log expected time. In *Automata, Languages and Programming, 22nd International Colloquium, ICALP95, Szeged, Hungary, July 10-14, 1995, Proceedings*, pages 159–170, 1995.

[NS17] Danupon Nanongkai and Thatchaphol Saranurak. Dynamic spanning forest with worst-case update time: adaptive, las vegas, and o($n^{1/2 - \epsilon}$)-time. In *Proceedings of the 49th Annual ACM SIGACT Symposium on Theory of Computing, STOC 2017, Montreal, QC, Canada, June 19-23, 2017*, pages 1122–1129, 2017.

[O'C09] 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.

[OP09] Tore Opsahl and Pietro Panzarasa. Clustering in weighted networks. *Social Networks*, 31(2):155–163, 2009.

[OR97] Chao-Wei Ou and Sanjay Ranka. Parallel incremental graph partitioning. *IEEE Trans. Parallel Distrib. Syst.*, 8(8):884–896, 1997.

[Par15] Merav Parter. Dual failure resilient BFS structure. In *Proceedings of the 2015 ACM Symposium on Principles of Distributed Computing, PODC 2015, Donostia-San Sebastián, Spain, July 21 - 23, 2015*, pages 481–490, 2015.

[Paw89] Shaunak Pawagi. A parallel algorithm for multiple updates of minimum spanning trees. In *Proceedings of the International Conference on Parallel Processing, ICPP '89, The Pennsylvania State University, University Park, PA, USA, August 1989. Volume 3: Algorithms and Applications.*, pages 9–15, 1989.

[PD06] Mihai Patrascu and Erik D. Demaine. Logarithmic lower bounds in the cell-probe model. *SIAM J. Comput.*, 35(4):932–963, 2006.

[Pel00] David Peleg. *Distributed Computing: A Locality-sensitive Approach*. Society for Industrial and Applied Mathematics, Philadelphia, PA, USA, 2000.

[PGT11] Konstantinos Patakakis, Loukas Georgiadis, and Vasileios A. Tatsis. Dynamic dominators in practice. In *15th Panhellenic Conference on Informatics, PCI 2011, Kastoria, Greece, September 30 - October 2, 2011*, pages 100–104, 2011.

[PK93] Shaunak Pawagi and Owen Kaser. Optimal parallel algorithms for multiple updates of minimum spanning trees. *Algorithmica*, 9(4):357–381, 1993.

[PK06] David J. Pearce and Paul H. J. Kelly. A dynamic topological sort algorithm for directed acyclic graphs. *ACM Journal of Experimental Algorithmics*, 11, 2006.

[Pou96] Han La Poutré. Lower bounds for the unionfind and the splitfind problem on pointer machines. *Journal of Computer and System Sciences*, 52(1):87 – 99, 1996.

[PP13] Merav Parter and David Peleg. Sparse fault-tolerant BFS trees. In *Algorithms - ESA 2013 - 21st Annual European Symposium, Sophia Antipolis, France, September 2-4, 2013. Proceedings*, pages 779–790, 2013.

[PPC02] Jung-Ho Park, Yoon-Young Park, and Sung-Hee Choi. Distributed algorithms solving the updating problems. *Korean Journal of Computational & Applied Mathematics*, 9(2):437–450, May 2002.

[PR86] Shaunak Pawagi and I. V. Ramakrishnan. An o(log n) algorithm for parallel update of minimum spanning trees. *Inf. Process. Lett.*, 22(5):223–229, 1986.

[PT07] Mihai Patrascu and Mikkel Thorup. Planning for fast connectivity updates. In *48th Annual IEEE Symposium on Foundations of Computer Science (FOCS 2007), Proceedings*, pages 263–271, 2007.

[Rei85] John H. Reif. Depth-first search is inherently sequential. *Inf. Process. Lett.*, 20(5):229–234, 1985.

[Rei87] John H. Reif. A topological approach to dynamic graph connectivity. *Inf. Process. Lett.*, 25(1):65–70, 1987.

[RIF02] Matei Ripeanu, Adriana Iamnitchi, and Ian T. Foster. Mapping the gnutella network. *IEEE Internet Computing*, 6(1):50–57, 2002.

[Rod08] Liam Roditty. A faster and simpler fully dynamic transitive closure. *ACM Trans. Algorithms*, 4(1):6:1–6:16, 2008.

[Rod13] Liam Roditty. Decremental maintenance of strongly connected components. In *Proceedings of the Twenty-Fourth Annual ACM-SIAM Symposium on Discrete Algorithms, SODA 2013, New Orleans, Louisiana, USA, January 6-8, 2013*, pages 1143–1150, 2013.

[RT07] Celso C. Ribeiro and Rodrigo F. Toso. Experimental analysis of algorithms for updating minimum spanning trees on graphs subject to changes on edge weights. In *Experimental Algorithms, 6th International Workshop, WEA 2007, Rome, Italy, June 6-8, 2007, Proceedings*, pages 393–405, 2007.

[Ruh03] Jan Matthias Ruhl. *Efficient Algorithms for New Computational Models*. PhD thesis, Department of Computer Science, MIT, Cambridge, MA, 2003.

[RV92] K.V.S Ramarao and S Venkatesan. On finding and updating shortest paths distributively. *Journal of Algorithms*, 13(2):235 – 257, 1992.

[RZ08] Liam Roditty and Uri Zwick. Improved dynamic reachability algorithms for directed graphs. *SIAM J. Comput.*, 37(5):1455–1471, 2008.

[RZ11] Liam Roditty and Uri Zwick. On dynamic shortest paths problems. *Algorithmica*, 61(2):389–401, 2011.

[RZ16] Liam Roditty and Uri Zwick. A fully dynamic reachability algorithm for directed graphs with an almost linear update time. *SIAM J. Comput.*, 45(3):712–733, 2016.

[San04] Piotr Sankowski. Dynamic transitive closure via dynamic matrix inverse (extended abstract). In *45th Symposium on Foundations of Computer Science (FOCS 2004), 17-19 October 2004, Rome, Italy, Proceedings*, pages 509–517, 2004.

[San05] Piotr Sankowski. Subquadratic algorithm for dynamic shortest distances. In *Computing and Combinatorics, 11th Annual International Conference, COCOON 2005, Kunming, China, August 16-29, 2005, Proceedings*, pages 461–470, 2005.

[SG89] John Michael Spinelli and Robert G. Gallager. Event driven topology broadcast without sequence numbers. *IEEE Trans. Communications*, 37(5):468–474, 1989.

[SG98] Bala Swaminathan and Kenneth J. Goldman. An incremental distributed algorithm for computing biconnected components in dynamic graphs. *Algorithmica*, 22(3):305–329, 1998.

[SI89] Mohan B. Sharma and S. Sitharama Iyengar. An efficient distributed depth-first-search algorithm. *Inf. Process. Lett.*, 32(4):183–186, 1989.

[Sib01] J.F. Sibeyn. *Depth First Search on Random Graphs*, volume 6 of *Report* -. Department of Computing Science, Ume University, 2001.

[SL93] X. Shen and W. Liang. A parallel algorithm for multiple edge updates of minimum spanning trees. In *Proceedings Seventh International Parallel Processing Symposium*, pages 310–317, Apr 1993.

[Smi86] Justin R. Smith. Parallel algorithms for depth-first searches i. planar graphs. *SIAM J. Comput.*, 15(3):814–830, 1986.

[Sre95] Vugranam C. Sreedhar. *Efficient Program Analysis Using DJ Graphs*. PhD thesis, McGill University, Montreal, Que., Canada, Canada, 1995. UMI Order No. GAXNN-08158.

[SS07] Dominik Schultes and Peter Sanders. Dynamic highway-node routing. In *Experimental Algorithms, 6th International Workshop, WEA 2007, Rome, Italy, June 6-8, 2007, Proceedings*, pages 66–79, 2007.

[ST83] Daniel Dominic Sleator and Robert Endre Tarjan. A data structure for dynamic trees. *J. Comput. Syst. Sci.*, 26(3):362–391, 1983.

[SV88] Baruch Schieber and Uzi Vishkin. On finding lowest common ancestors: Simplification and parallelization. *SIAM J. Comput.*, 17(6):1253–1262, 1988.

[SW15] Xiaoming Sun and David P. Woodruff. Tight Bounds for Graph Problems in Insertion Streams. In *Approximation, Randomization, and Combinatorial Optimization. Algorithms and Techniques (APPROX/RANDOM 2015)*, volume 40 of *Leibniz International Proceedings in Informatics (LIPIcs)*, pages 435–448, 2015.

[Tar72] Robert Endre Tarjan. Depth-first search and linear graph algorithms. *SIAM J. Comput.*, 1(2):146–160, 1972.

[Tar74] Robert Endre Tarjan. Finding dominators in directed graphs. *SIAM J. Comput.*, 3(1):62–89, 1974.

[Tar75] Robert Endre Tarjan. Efficiency of a good but not linear set union algorithm. *J. ACM*, 22(2):215–225, April 1975.

[Tar79] Robert Endre Tarjan. A class of algorithms which require nonlinear time to maintain disjoint sets. *Journal of Computer and System Sciences*, 18(2):110 – 127, 1979.

[Tar97] Robert Endre Tarjan. Dynamic trees as search trees via euler tours, applied to the network simplex algorithm. *Mathematical Programming*, 78:169–177, 1997.

[Tho99] Mikkel Thorup. Decremental dynamic connectivity. *J. Algorithms*, 33(2):229–243, 1999.

[Tho00] Mikkel Thorup. Near-optimal fully-dynamic graph connectivity. In *Proceedings of the Thirty-Second Annual ACM Symposium on Theory of Computing, May 21-23, 2000, Portland, OR, USA*, pages 343–350, 2000.

[Tho04] Mikkel Thorup. Fully-dynamic all-pairs shortest paths: Faster and allowing negative cycles. In *Algorithm Theory - SWAT 2004, 9th Scandinavian Workshop on Algorithm Theory, Humlebaek, Denmark, July 8-10, 2004, Proceedings*, pages 384–396, 2004.

[Tho05] Mikkel Thorup. Worst-case update times for fully-dynamic all-pairs shortest paths. In *Proceedings of the 37th Annual ACM Symposium on Theory of Computing, STOC 2005*, pages 112–119, 2005.

[Tsi88] Yung H. Tsin. On handling vertex deletion in updating spanning trees. *Inf. Process. Lett.*, 27(4):167–168, 1988.

[Tsi02] Yung H. Tsin. Some remarks on distributed depth-first search. *Inf. Process. Lett.*, 82(4):173–178, 2002.

[TV84] Robert Endre Tarjan and Uzi Vishkin. Finding biconnected components and computing tree functions in logarithmic parallel time (extended summary). In *25th Annual Symposium on Foundations of Computer Science*, pages 12–20, 1984.

[TvL84] Robert Endre Tarjan and Jan van Leeuwen. Worst-case analysis of set union algorithms. *J. ACM*, 31(2):245–281, 1984.

[VD86] Peter J. Varman and Kshitij Doshi. A parallel vertex insertion algorithm for minimum spanning trees. In *Automata, Languages and Programming, 13th International Colloquium, ICALP86, Rennes, France, July 15-19, 1986, Proceedings*, pages 424–433, 1986.

[VMCG09] Bimal Viswanath, Alan Mislove, Meeyoung Cha, and Krishna P. Gummadi. On the evolution of user interaction in facebook. In *Proceedings of the 2nd ACM SIGCOMM Workshop on Social Networks (WOSN'09)*, August 2009.

[WT92] Jeffery Westbrook and Robert Endre Tarjan. Maintaining bridge-connected and biconnected components on-line. *Algorithmica*, 7(5&6):433–464, 1992.

[Wul13] Christian Wulff-Nilsen. Faster deterministic fully-dynamic graph connectivity. In *Proceedings of the Twenty-Fourth Annual ACM-SIAM Symposium on Discrete Algorithms, SODA 2013, New Orleans, Louisiana, USA, January 6-8, 2013*, pages 1757–1769, 2013.

[Wul17] Christian Wulff-Nilsen. Fully-dynamic minimum spanning forest with improved worst-case update time. In *Proceedings of the 49th Annual ACM SIGACT Symposium on Theory of Computing, STOC 2017, Montreal, QC, Canada, June 19-23, 2017*, pages 1130–1143, 2017.

[WY13] Oren Weimann and Raphael Yuster. Replacement paths and distance sensitivity oracles via fast matrix multiplication. *ACM Trans. Algorithms*, 9(2):14:1–14:13, March 2013.

[Yel93] Daniel M. Yellin. Speeding up dynamic transitive closure for bounded degree graphs. *Acta Informatica*, 30(4):369–384, 1993.

[Zar00] Christos D. Zaroliagis. Implementations and experimental studies of dynamic graph algorithms. In *Experimental Algorithmics, From Algorithm Design to Robust and Efficient Software [Dagstuhl seminar, September 2000]*, pages 229–278, 2000.

[Zha86] Y. Zhang. *Parallel algorithms for problems involving directed graphs.* PhD thesis, Drexel University, Philadelphia, PA, 1986.

[ZLMZ05] Beichuan Zhang, Raymond Liu, Daniel Massey, and Lixia Zhang. Collecting the internet as-level topology. *SIGCOMM Computer Communication Review*, 35(1):53–61, 2005.

Index