EFFICIENT MINING OF ACTIVE COMPONENTS IN A NETWORK OF TIME SERIES

MAHTA SHAFIEESABET

A THESIS SUBMITTED TO THE FACULTY OF GRADUATE STUDIES
IN PARTIAL FULFILMENT OF THE REQUIREMENTS
FOR THE DEGREE OF
MASTERS OF SCIENCE

GRADUATE PROGRAM IN ELECTRICAL ENGINEERING & COMPUTER SCIENCE
YORK UNIVERSITY
TORONTO, ONTARIO

JANUARY 2022

Abstract

Let a *network of time series* be a set of nodes assuming an underlying network structure, where each node is associated with a discrete-time time series. The road network, the human brain, online social media are a few examples of domain-specific applications that can be modelled as networks of time series. Now assume that the sequence of time series data points observed on a node determines whether the node is *on* (*active*) or *off* (*inactive*). Then, at each time step, a set of induced subgraphs can be formed from the subset of active nodes; we call these induced subgraphs *active components*. In this research, our goal is to efficiently detect and maintain/report the active components over time.

This setting can also be formalized using the concept of a dynamic graph. A *dynamic graph* G can be viewed as a discrete sequence of static graphs $G_1, G_2, ..., G_t$ where $G_t = (V_t, E_t)$ and t is the index of the time steps. Each G_i is a snapshot of G. In our setting of a network of time series, the original network is represented as a graph G that has a fixed topology and its nodes become active or inactive over time, depending on the time series values. At each time step, the induced graph defined by the set of active nodes is a snapshot G_i of G. This problem where the original graph is fixed and its nodes become on/off describes a specific

dynamic graph problem, known in the literature as the *subgraph model*.

To detect the sequence of active components at each step, it is enough to find the spanning forest induced by the active nodes, as each spanning tree in the spanning forest represents each of the active components. The naive approach to address the problem is to rely on a depth-first search computation on the original graph, at each time step. However, this approach might be prohibitive for large graphs. The main contribution of this thesis is that we propose and implement an efficient algorithm for detecting the spanning forest and therefore the sequence of active components, while avoiding a full re-computation on the original graph. We provide a theoretical analysis of the time complexity of the proposed algorithm and show that it improves the time complexity of the known state of the art algorithm by *a log factor*. We also empirically evaluate the running time performance of the proposed algorithm against state-of-the-art algorithms and other sensible baselines.

The ability to efficiently detect active components in real-time can inform various critical situations in domain-specific applications, such as monitoring specific traffic conditions in the road network, monitoring critical human brain activity, or detecting persistent social network discussions, to name a few.

Acknowledgements

I would like to give my special thanks to my supervisor, Dr. Manos Papagelis; it would not have been possible to do this research without his help and support.

I would like to extend my special thanks to my thesis oral exam committee members, Dr Gene Cheung and Dr Mojgan Jadidi.

I would like to thank my auntie Nahid and her family for all of their love, help and support and for being my second family when I was far from my family in Iran.

I would also want to thank my loving parents and siblings (Azadeh, Mahdyeh, Azin, Mohammad) for keeping me in their hearts and their emotional support for me from a long-distance away.

And at the end, I would like to thank my lovely friends Niloofar Radgoudarzi (rad), Kenneth Tjhia, Neda Shokraneh, Farnoosh Javadi, Sheyda Zarandi, Alireza Naeiji, Fazel Arasteh, Xeleb Nadri, Parsa Farshadfar, Ramin Shirali, and all of my other dear friends who added fun to my journey and never let me be down.

Table of Contents

List of Figures

List of Tables

Chapter 1

Introduction

Let a *network of time series* be a set of nodes assuming an underlying network structure, where each node is associated with a discrete-time time series. The road network, the human brain, online social media are a few examples of domain-specific applications that can be modelled as networks of time series. Now assume that the sequence of data points observed on a node determines whether the node is *on* (*active*) or *off* (*inactive*). Then, at each time step, the active nodes form a set of induced subgraphs that we call *active components*. The aforementioned setting can also be formalized using the concept of a dynamic graph.

A *dynamic graph* is a graph that is changing over time like G as $G = (G1, G2, ..., G_t)$ where $G_t = (V_t, E_t)$ and t is the number of time steps. Each G_i is also known as a snapshot of G. In the setting that we described earlier, where the input graph is fixed, and nodes become active or inactive through time, each snapshot is an induced subgraph of the input graph to active nodes. This setting is one of the dynamic graphs problems known as *subgraph*

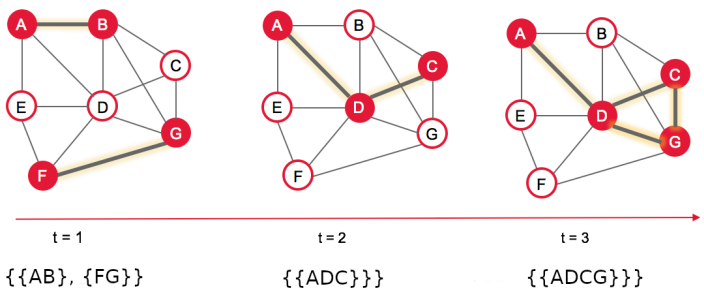


Figure 1.1: Set of active components at timesteps t = {1, 2, 3}

model. Fig. 1.1 provides an illustrative example of such active components for a small graph and a short period of three-time steps. In the figure, the nodes of the graph $\{A, B, ..., G\}$ are observing time series with three time-steps that determine their state (active or inactive), and at each time step, the set of active nodes (shown in red) forms active components (subgraphs composed of red nodes and highlighted edges).

Previous works maintained a data structure that can answer a query about two vertices if they are in the same connected component or not in a certain amount of time. They were trying to reduce this time as well as making a smaller data structure [15, 18]. On the other hand, we maintain the spanning forest and report each active components members. Some previous works maintain the DFS-tree (forest), which is also a spanning tree (forest), but our algorithm focuses on maintaining a spanning tree so is simpler and faster in terms of time complexity [33, 4].

In this research, we present an efficient algorithm for detecting the spanning forest induced to active nodes and, as a result, the sequence of active components while avoiding a full

re-computation on the original graph. To approach the problem of interest, we use and modify an existing data structure and algorithm for maintaining a DFS-tree in a dynamic graph under vertex update. The work is done by Baswana et al. [7]. Their work is theoretical, but we implemented their algorithm and the algorithm with our modification. We make their time complexity faster by a log factor. The main idea is that they are maintaining a DFS-tree (DFS-forest), but we want a spanning tree (spanning forest) to get rid of the computations that make sure the tree is a DFS-tree.

As we mentioned earlier, traditionally, in the fully dynamic problem setting, we care about *dynamic connectivity*, where we seek quick answers to queries of the form "are nodes x and y connected in the updated graph?". On the other hand, our setting focuses on maintaining the active components' network structure, allowing for rich meta-analysis and modelling of related domain-specific phenomena. For example, we can use the sequence of active components to answer questions such as: what active components look like, how persistent they are over time, how their network structure changes over time.

1.1 Motivating Applications

The road network, the human brain, sensor networks, online social media are a few examples of domain-specific applications that can be modelled as networks of time series. The ability to efficiently detect active components in real-time can inform various critical situations in domain-specific applications, such as specific road network conditions, critical human brain activity, or persistent social media discussions, to name a few. Below we elaborate more on

such examples and how they can be modelled as networks of time series.

1.1.1 Transportation Networks

Transportation networks consist of sets of connected road segments that intersect, creating road intersections. Each intersection may generate time-series in a transportation network by monitoring traffic conditions at that intersection over time. To formalize transportation networks in our context, they must be modelled into a graph representation. The most well-known methodology of modelling transportation networks is by representing road intersections as nodes and road segments as edges. Each node is generating time series that turn that node active or inactive. An active node represents a congested intersection, while an inactive node represents an uncongested intersection. Therefore, finding active components over time is equivalent to finding the set of congested roads across the network at each time step. We can use the computed set of active paths in traffic prediction through real-time forecasting of traffic information [31].

1.1.2 Human Brain

The human brain is divided into distinct but inter-connected regions. In an fMRI scan, each brain region generates a time series that monitors brain activity over time. To model the human brain as a graph, distinct brain regions can be represented as nodes. Each node turns active or inactive based on the data extracted from the time-series at each time step. An active node represents an active brain region, while an inactive node represents an inactive

region. Active components represent a set of interconnected brain regions that are highly active at each time step. Finding the set of active components outlines correlations between different brain regions over time [23].

1.1.3 Sensor Networks

A sensor network is a collection of connected and spatially distributed sensors that collectively sense physical and environmental properties. Each sensor is connected to a time series that continuously generates sensor data. For instance, each sensor generates a time series of temperature measurements in a smart building setting. A sensor network can be modelled into a graph representation where each sensor is a node, and the link between two neighbouring sensors is an edge. Nodes can turn active/inactive over time based on the time series values. In one scenario, a node turns active whenever an anomaly is detected at a specific time step, such as an unusually high or low temperature. In another scenario, a node turns active whenever a sensor measurement passes a specific threshold, such as a previously defined temperature value. When two or more connected nodes turn active, an active component is constructed in the graph. Finding the set of active components finds the underlying relationship between the sensors included in the path [8].

1.1.4 Online Social Networks

An online social network can be modelled into a graph representation of co-evolving time series. In Twitter, hashtags can be represented by graph nodes and the relationship between

each hashtag is represented by edges linking them. Each hashtag generates a time series of live tweets over time. Nodes turn active/inactive based on the trending score of each hashtag. An active node represents a trending hashtag. When two or more connected hashtags are trending, they form an active component. Finding the set of active components over time defines how two hashtags are related to each other, which is critical in economic, social, and political analysis using tweets as a data set. Or may be of interest to determine connected groups of people all meeting some common criteria, for example a group of friends on a social media platform all of whom attending some event.

1.2 Contributions

In summary, the major contributions of this work are the following:

- We introduce the novel problem of detecting active components in a network of time series. An active component is defined as the induced subgraph of a set of active nodes.
- We propose *ActiveComp, an efficient algorithm for addressing the problem of interest in the subgraph model setting. In this setting, we are given a graph of fixed topology (fixed nodes and edges), while the nodes in the graph become on (active) and off (inactive) over time. At each time step, a set of active components is detected and reported.*
- *We provide a theoretical analysis of the time complexity of the proposed algorithm and show that it improves the currently known state of the art by a log factor.*
- *We empirically compare the running time performance of the proposed algorithm and*

demonstrate that it outperforms the state-of-the-art and other sensible baseline algorithms, for a varying number of parameters.

- *While there are many advances in theoretical analysis of dynamic graph algorithms, there is a gap between theoretical and empirical studies [21]. The problem with theoretical results is that while the worst-case or amortized time complexity performance of the algorithm is known, one can barely say much about their expected performance in a real-world application or setting. Our contribution is therefore to provide better insights and understanding of the performance and utility of the proposed algorithms in specific scenarios, through implementation and empirical evaluation of the proposed method against sensible alternatives.*

1.3 Thesis Organization

The remainder of this thesis is organized as follows: Chapter 2 reviews the related work to our research. The technical problem of interest in this research is presented in Chapter 3. Chapter 4 introduces preliminaries of the problem and discusses the current state-of-the-art solutions to our problem. Our proposed methods, the overall algorithmic analysis and implementation details are also presented in Chapter 4. Chapter 5 presents an experimental evaluation of our proposed methods against the state-of-the-art algorithm and other sensible baselines. We conclude the work and provide directions of future work in Chapter 6.

Chapter 2

Related Work

We can categorize related works into three most important sections: *streaming graph mining*, *time series mining*, and, *dynamic graph connectivity*.

2.1 Streaming Graph Mining

There are many studies on evolving graphs. To handle giving an evolving graph as an input to an algorithm, we can feed it as a data input stream. The data can be a stream of tuples like (e, s) where e is an edge and $s \in \{0, 1\}$ and $(e, 0)$ means we want to delete e from and $(e, 1)$ means we want to insert e to the evolving graph. Usually, in graph stream algorithms, the input is a stream of edges, while in our problem, nodes are changing, and we feed the change as a binary vector of size of the graph where each element shows the status of a node in the input graph.

McGregor [32] presented a survey of graph stream algorithms. Anagnostopoulos et al.

[3] presented an algorithm for estimating clusters in an evolving graph, assuming that the graph is generated by evolving stochastic block model; an extension to the stochastic block model [1] to the evolving setting. Rannou et al. [38] proposed an approximation method for computing strongly connected components in stream graphs. They also implemented and assessed existing algorithms with polynomial (based on the number of the nodes) time complexity in this area. Yang et al. [48] proposed a model for capturing the most frequently changing component (MFCC) in a streaming graph.

2.2 Time Series Mining

Time-series mining problems are problems where the input is dependent on time, and we have a series of inputs through time. In work by Cai et al., [10], we observe that coevolving series applications often have an (implicit or explicit) network structure that partially encodes the dependencies (for example, traffic measurements at intersections (nodes) in a road (edge) network). The authors refer to such a set of coevolving time series and an associated network as a network of coevolving time series. As we mentioned in the introduction, we can also formulate our problem as a time series network when G_t refers to the input graph induced to active nodes at time step t.

Cai et al. [10] proposed an algorithm based on matrix factorization to infer the missing values from the input time series called Dynamic Contextual Matrix Factorization (DCMF); their algorithm is effective, especially when there are lots of missing values. Coevolving time series are well studied and find a wide range of applications in any problem domain in

different sciences which can be modeled as a graph [30], including computer or social network analysis [14], sensor (biological, environmental, etc.) network monitoring [49], and financial data analysis [50] (these application references are found in [36]).

2.3 Dynamic Graphs

Dynamic graph algorithms are the ones that have a data structure that supports edge/vertex insertion or deletion. The most popular graph problems in dynamic graph studies are connectivity, reachability, shortest path, and matching [21]. Our problem is more closely related to the dynamic connectivity problem. We have an initial static graph, and we can preprocess it and use the information in the preprocessing time when the vertices become inactive/active to find the active components. In this setting that we have an initial static graph, and nodes become on/off, the connectivity problem is called *subgraph model*. Based on a recent paper by Hanauer et al.[21] a survey of recent papers and experimental results on fully dynamic graph algorithms, we can categorize the related dynamic graph papers to our work in the following four sections: *Fully Dynamic Connectivity*, *Dynamic DFS*, *Connectivity in the Subgraph Model*, *Emergency Planning*.

2.3.1 Fully Dynamic Connectivity

In fully dynamic connectivity problems, we have an evolving graph that accepts a batch of updates (a set of edge insertion and deletion). After each batch, we want to answer *connectivity queries* - queries that inquire whether two nodes are connected. However, in

our research, we also get a set of updates, but instead of answering queries, we maintain the spanning forest between the active nodes.

Henzinger and King [25] presented the first fully dynamic algorithm, which maintains approximate spanning tree, connectivity and bipartiteness in polylogarithmic time per edge insertion or deletion. Their algorithm was randomized. They also proposed a data structure, called the Euler tour tree data structure, which has since been utilized in many subsequent works. Two years later, Holm et al. [26] proposed the first deterministic fully dynamic algorithm with polylogarithmic time per operation (edge insertion or deletion). The currently fastest fully dynamic algorithm is by Huang et al. [27]. Alberts et al. [2] and Iyer et al. both provided experimental studies on fully dynamic connectivity. Besides implementing algorithms and comparing them on different datasets, they speed them up by some heuristics.

2.3.2 Dynamic DFS

Depth First Search (DFS) is a fundamental traversing graph search algorithm that is the basis of many other graph algorithms like finding strongly connected components, checking whether a graph is planar, and finding biconnected components in a graph. We can show the DFS algorithm's result by a DFS-tree [47]. Maintaining the DFS-tree in a dynamic graph is a well-studied problem. There is an important relationship between these papers and our paper, and that is, they all work with node updates; besides, a DFS-tree is also a spanning tree. Following, we elaborate on the most important developments of this problem.

Baswana et al. [4] provided a framework and algorithms for fault-tolerant, fully dynamic,

and incremental DFS-tree problems. Their idea was to use disjoint tree partitioning of the DFS-tree. Then Chen et al. [12] improved their result using their framework and the celebrated fractional cascading technique. Then three years later, Baswana et al. and Nakamura et al. [5, 33] presented algorithms for maintaining a fully dynamic DFS-tree (insertion and deletion of vertices or edges) in undirected graphs, which has the best results till now. Later on, Baswana et al. [7] simplified their algorithm for fault-tolerant problems and made it more efficient as well. Our proposed method utilizes the results of Baswana et al. [7] for the subgraph model. Note that in our setting, the original graph is fixed; therefore, further optimizations are possible. We further elaborate on these in Section 4.3. Briefly, at each time step, we want to find the spanning forest, so we can assume some vertices fail at each step. Using the same data structure, we modified their algorithm and made it faster to maintain a spanning tree instead of a DFS-tree.

Yang et al. [47] provided a framework and algorithms for maintaining DFS-tree undergoing updates (edge insertion or deletion) in general directed graphs. Khan et al. [29] worked on dynamic DFS in semi streaming model. Finally, Nakamura et al. [34] presented a space-efficient algorithm. At the same time, it is slower than the other algorithms; thus, it is useful for cases that space is essential or we cannot have significant data structures.

2.3.3 Connectivity in the Subgraph Model

As we also mentioned in the introduction, a dynamic graph model that nodes are switched on and off is called Subgraph Model. Previous works mostly answer connectivity queries

in a subgraph model where a query asks whether two vertices are connected. However, our research is also a subgraph model, except we try to maintain a spanning forest after each time step where some nodes' status is switched in that time step. Duan and Ran [15] studied subgraph connectivity in subgraph model under vertex update for the first time. They presented the first subgraph connectivity structure to answer connectivity queries. A year after Chan et al. [11] greatly improved Duan's paper's result. Duan and Pettie [16] presented a data structure that can answer connectivity queries if there is a path between two nodes after deleting d vertices in time linear to d. Their connectivity structure update time is polynomial in d. They presented a better data structure in [17], which is dramatically better in every measure of efficiency except for construction time. They improved their result again [18]. Borradaile et al. [19] and Frigioni et al. [9] provided connectivity oracles for planar graphs. Baswana et al. [6] presented an algorithm that maintains strongly connected components after the failure of a set of edges or vertices.

2.3.4 Emergency Planning

Emergency planning problems are similar to subgraph model problems (our setting), except in these problems, they try to implement an algorithm that is optimized for the case an emergency happens. So they have a single batch of updates and want to answer the queries after that batch in case of an emergency instead of a sequence of updates (like our research) and queries [24]. Thus their data structure is optimized only for one-time use, and they have to reconstruct it after each emergency.

Henzinger and Neumann [24] showed how you could transform a connectivity oracle subject to a d-vertex failure to a fully dynamic algorithm subject to d-vertex updates for emergency planning, where vertices can fail and recover individually. Patrascu and Thorup [37] proposed the Emergency Planning problem about answering the connectivity queries quickly after a batch of updates. In this paper, given a set of d edges update, they provide an algorithm that can obtain the number of connected components and size of them and a structure for answering connectivity queries in time $O(d \operatorname{polylog} n)$.

Chapter 3

Problem Definition

Suppose we have an undirected graph where each node is associated with a time series. The time series can be independent (or not); we do not make any assumption. The network topology of the graph is fixed and given. If a node's time series satisfies some prescribed property at a time t, we say this node is *active* at time t (for example its value is above a threshold θ), otherwise we say this node is *inactive* at time t. We say that an edge at time t is active if the nodes at its endpoints are active at time t. We define an *active component* at time t if it is an induced graph of a set of active nodes at time t. Our problem is determining the set of active components at time t. We can determine the active components by running a DFS algorithm on the graph at each time step. However, we do not want to traverse the whole graph at each time step, especially when the number of nodes that switched their status is not a lot or in another scenario when only a few nodes are active.

In this thesis, we focus specifically on a connected, undirected graph where each time

Table 3.1: Summary of Notations

Symbol	Definitions and Descriptions
G	undirected graph
T	DFS-tree of G
V	set of vertices
E	set of edges
X_t^i	value of the ith node's time series at time t
V_{on}^t	set of active nodes at time t
G_{on}^t	subgraph of G induced by V_{on}^t
V_{off}^t	set of inactive nodes at time t
G_{off}^t	subgraph of G induced by V_{off}^t
$\vec{X}_t$	vector of all time series' values at time t
$Time$	index set for the time series'
Γ	range of the time series' values of nodes
γ	subset of Γ that defines an active node
C_t	set of active components at time t

series is a binary data stream. In this setting, we call a node active at time t if its associated time series has the value one at time t.

In the following, we introduce definitions for the terms we use, and at the end, we formulate the problem statement.

Definition 1 (Network of Time Series)**.** *A graph $G = (V, E)$, where each node i, $i = 1, 2, \ldots, n$, is associated with a time series $X^i = \{X_t^i : t \in T, X_t^i \in \Gamma\}$, where Time is an index set and Γ the set of values which any node at any time index of the time series may take.*

Definition 2 (Active Node)**.** *A node i is active at time t if $X_t^i \in \gamma \subset \Gamma$. When this property is not satisfied we call the node inactive. Also the notations V_{on}^t and V_{off}^t are the set of active nodes and inactive nodes respectively at time step t. In this thesis we focus on binary times series, where $X_t^i \in 0, 1$, and by convention we assume that $X_t^i = 0$ represents an inactive node at time t, and $X_t^i = 1$ represents an active node at time t.*

Definition 3 (Active component)**.** *A set of nodes $C \subseteq V$ is an active component at time t if they are all active at this time and also form a connected component. Otherwise, if we assume that G_{on}^t is the subgraph of G induced by V_{on}^t then an active component is a connected component of G_{on}^t.*

We are now in position to formally define the problem of interest in this work.

Problem 1 (Finding Active components)**.** *Given a network of time series* $G = (V, E)$, $Time = \mathbb{Z}^+$, *as well* $\vec{X}_{t-1}$ *and* $\vec{X}_t$, *where,* $\vec{X}_t$ *is a vector of time series values at time* t *where the* i*th component is the time series value associated with node* i*; we want to determine* C_t *which is the sets of active components at time* t*, or in other words we want to find all the connected components of* G^t_{on} *for all* $t \in Time$.

We solve a particular instance of Problem 1 where $G = (V, E)$ is connected, $Time = \{1, 2, \ldots, T_c\}$ where T_c is a constant, $\Gamma = \{0, 1\}$, and $\gamma = \{1\}$.

Table 3.1 provides a summary of notations that we use in this thesis.

Chapter 4

Methodology

In this chapter, we first go through the problem's setting, input and output, in more detail. Then, in the first section, we go through some preliminaries for our algorithm. Then in the second section, we introduce our baseline method, a Simple DFS. In the third section, we present our algorithm and compute the algorithm's time complexity in the last section.

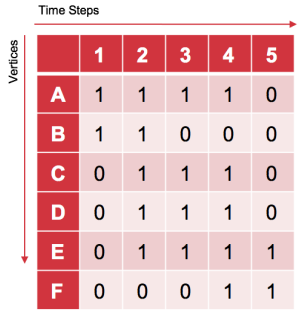
Algorithm 1 shows the big picture of one experiment simulation of the problem in our research. The inputs of Algorithm1 are as follows:

- Set $Time$: This is index set for time series which in this thesis is $\{1, 2, \ldots, T_c\}$.

- Graph $G = (V, E)$: A static undirected graph G is given as an adjacency list and stored in memory during run-time.

- Matrix X : A $|V| \times |T_c|$ matrix contains information about which nodes are active/inactive at each time step. So X_t is the t'th column of this matrix. Figure 4.1 depicts a sample of matrix X. As shown in Figure 4.1, rows represent nodes while columns represent time steps.

For example, the entry at row = 3 and column = 2 means that node A is active at time step 2. This matrix has been generated by a synthetic data generator, which will be discussed more in depth in section 5.1.2.



	1	2	3	4	5
A	1	1	1	1	0
B	1	1	0	0	0
C	0	1	1	1	0
D	0	1	1	1	0
E	0	1	1	1	1
F	0	0	0	1	1

Figure 4.1: Sample input file. Contains information about nodes turning active/inactive. Looking at entry row = B and column = 2, we know that node B is active at $t = 2$

Algorithm 1 `Active_Comp_Through_Time`$(Time, G, X)$

Input: Static undirected graph G, Nodes' status matrix X, Time series' index set $Time$
Ensure: Computed $C = \{C_t | t \in Time\}$
  for all $t \in Time$ **do**
    `Active_Comp`(G, X_{t-1}, X_t)
  end for

The output of this algorithm is $C = \{C_t | t \in Time\}$ while C_t is the set of active components at time t as it is in the problem statement.

In Algorithm 1 we loop over time. At each time step, we call the function `Active_Comp` which computes what we desire in the problem statement. This function is not defined with this name in our research; in fact, it can be replaced by a proper method like the baseline algorithm we introduce in section 4.2 or our algorithm in section 4.3.

4.1 Preliminaries

4.1.1 DFS-tree

Let G be an undirected graph. If we run a Depth First Search on G, the edges that we traverse in this search form a rooted tree, a spanning tree of G that we call a DFS-tree. In other words, in Algorithm 2 which is a recursive function that computes the DFS-tree T rooted at r, edges that are being added to T in line 4 form T and are called tree-edges. All other non-tree edges are back-edges. A back edge is an edge that is between a node and one of its ancestors. Here again we add a dummy node to G which is connected to all of the nodes in G and is always active and we start the DFS traversal search on the dummy node.

Algorithm 2 `Compute_DFS-tree(`G`,`T`,`r`)`

Input: Static undirected graph G, tree T, starting node r
Ensure: Computed DFS-tree T of G rooted at the dummy node

```
T.adjList.push_back(r)
r.visited ← true
for all ngbr ∈ r.neighbours do
    if ngbr.visited = false then
        add edge (r, ngbr) to T
        remove ngbr from r.neighbours
    end if
    Compute_DFS-tree(G,T,ngbr)
end for
```

An important **property** of DFS-trees is that we can't have a cross edge as a non-tree edge of a DFS-tree. A cross edge is an edge that is neither tree-edge nor back-edge. This property is cused by the nature of depth first traversal.

Proof. In depth-first traversal search, when we visit a node, we visit its neighbours and then neighbours of the neighbours until we can't go in-depth anymore. Using proof by contradiction, assume there is a cross edge (a, b) as a non-tree edge in a DFS-tree and we visit a earlier than

b. In a DFS procedure after visiting a, we visit b as a descendant of a; otherwise, it contradicts the DFS procedure. We also say that a and b have ancestor-descendant relation. □

For instance, in figure 4.2 we can see an undirected graph on the left and its DFS-tree rooted at node 1 on the right. In Figure4.2b tree edges are solid, and non-tree edges are dashed. As we can see, all the non-tree edges are back edges.

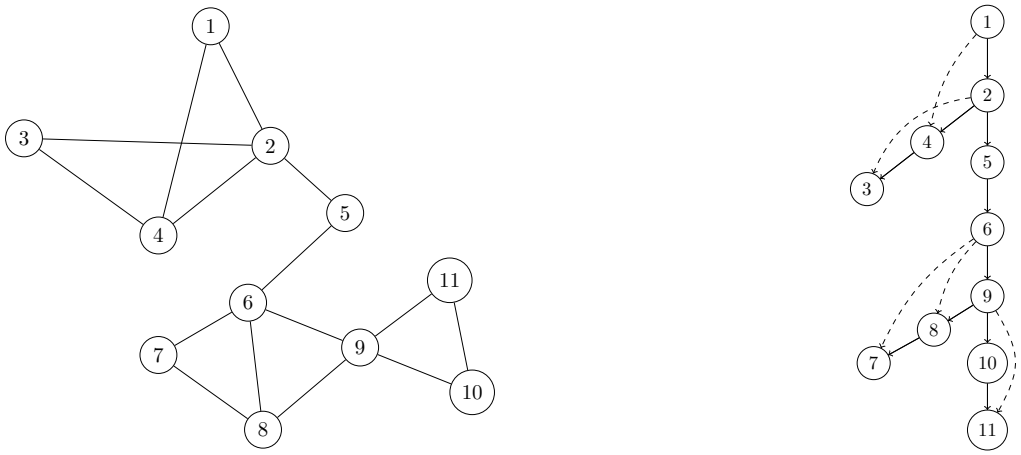


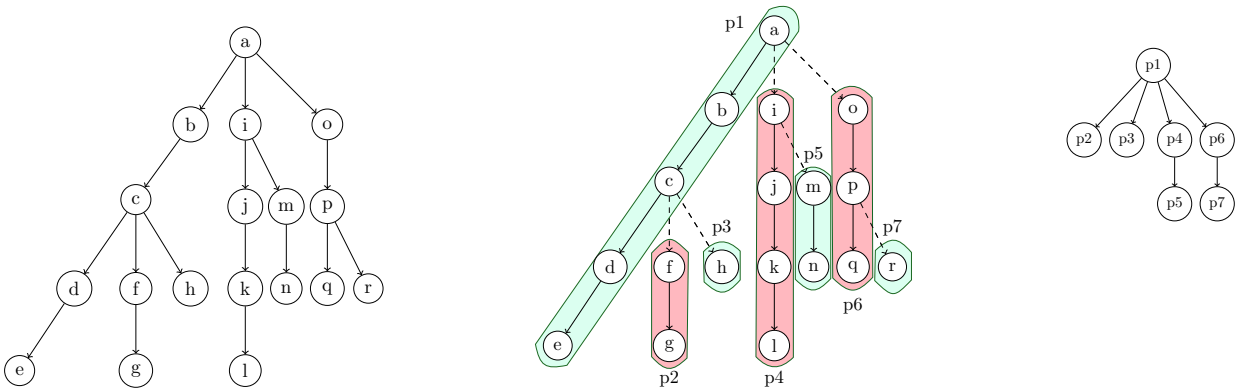
Figure 4.2: An undirected graph and its DFS-tree rooted at node 1

4.1.2 Heavy-Light Decomposition & Shallow Tree Representation

Heavy-Light Decomposition [41] is a technique for partitioning a rooted tree into a set of paths. These paths themselves form a rooted tree (called shallow tree) with a maximum height of $\log(n)$ while n is the number of nodes in the original tree . We will see how useful these decomposition and representation are later in the algorithm. Still, intuitively, this decomposition helps us answer queries like finding the lowest common ancestor of two nodes in a tree more efficiently.

Given a rooted tree for each node, we mark the edge to the heaviest child as solid and the

edges to the other children as dashed. The heaviest child is the one in which the number of vertices in the subtree rooted at that child is the biggest among its siblings. If we only look at solid edges, they form a set of vertex-disjoint paths connected hierarchically with dashed edges. Figure 4.3a shows a rooted tree. Figure 4.3b shows the same tree after marking edges solid or dashed and how paths are defined between the solid edges. If we capsule all the nodes of a path in a super-node in Figure 4.3c we can observe that dashed edges of the tree and these super-nodes form the shallow tree representation.



(a) (b) (c)

Figure 4.3: A figure that contains three subfigures

In the implementation, a shallow tree is a set of paths which we call each of them a `nodePath` and in each `nodePath` we save a pointer to its parent called *par* in the shaloow tree as well as two pointers to the start node and end node of the path. In the next section we'll see why we only save start node and end node of a path. For each node in the graph we save a pointer to the `nodePath` it belongs to.

Lemma 1. *(Baswana et al.[7]) The height of a shallow tree ST of a dfs-tree T is less than* $\log(n)$*. It means the hight of any node in a shallow tree is less than* $\log(n)$*.*

Proof. Every node in ST is a dashed edge in T. Thus the maximum height of ST is equal to the maximum number of dashed edges in root to leaves paths in T. For every dashed edge

(u, v) while u is v's parent, the size of subtree rooted at v is less than half of the size of the subtree rooted at u. Thus on any root to leaf path in T we maximum can have $\log(n)$ dashed eedges. Therefore, the height of any node in ST is less than $\log(n)$. □

Lemma 2. *(Baswana et al.[7]) There is a non-tree edge between two nodes from two different* `nodePaths` *if and only if one of them is the ancestor of the other.*

Proof. Assume (u, v) is a non-tree edge of T and p_u, p_v are their corresponding `nodePaths` in ST. If (p_u, p_v) is a cross edge in ST because edges in ST are some edges in T, (u, v) is a cross edge in T. Thus as we don't have any cross edge in T, p_u and p_v have a ancestor-descendant relastion. □

4.1.3 DFS-tree Enumeration

Given a rooted DFS-tree T, we can perform a DFS starting at its root and index every node as the time it was being visited by this search. We call a node's index the node's `dfs_id`. In this way the first node we visit in the DFS is the root of T and has a `dfs_id` of 1, and the last node has a `dfs_id` of n, which n is the number of nodes in the T. After visiting a node in a DFS on T, we randomly visit one of its children who is not visited. However, here in this work, after visiting a node, we visit its heaviest child that is not visited yet. For calculating the heaviest child, we run a DFS recursively on T, and for each node, we calculate and save the size of its subtree as a variable called `sizeOfsubT`.

As an example, let us index nodes of the tree in Figure 4.3a by performing a DFS on it. The ordering we get at the end is shown in the list below. Each tuple includes a node label and the node `dfs_id`, and the tuples are arranged from left to right in an increasing order based on their `dfs_id`s. We call this list `dfsOrderedList`.

$$(a, 1), (b, 2), (c, 3), (d, 4), (e, 5), (f, 6), (g, 7), (h, 8), (i, 9), (j, 10), (k, 11), (l, 12),$$

$$(m, 13), (n, 14), (o, 15), (p, 16), (q, 17), (r, 18)$$

If we look at the paths generated by Heavy-Light Decomposition of a DFS-tree and its dfs-ordered list, we can see each path is an interval in the dfs-ordered list. Again, if we look at the dfs-ordered list of the tree in Figure 4.3a we can show it as a list of all of its paths as follows:

$$p1, p2, p3, p4, p5, p6, p7$$

As a result, we can see nodes on a path have consecutive `dfs_id`s. This fact is useful for implementation, so for extracting the paths of T we can use its dfs-ordered list and also for saving each path, all we need to save is the a pointer to start node and to the end node of the path. Start node in a `nodePath` is the node with smaller `dfs_id` between two end points and is closer to the root of T. For example for `nodePath` $p4$ in Figure 4.3b we save a pointer to node i as $p4$ start node and a pointer to node l as it's end point.

4.2 Simple DFS (Baseline Method)

At each time step, given a static undirected graph $G = (E, V)$ and information about which nodes are active/inactive at each time step X_t,X_{t-1}, the baseline method provides a naive solution to our problem. The baseline method is inefficient and not scalable to massive graphs despite giving a solution. The input and the output of all algorithms are the same.

At each time step first, we update the status of every node based on X_t, X_{t-1} where the status of a node is a feature called *active* in class *node*. Then we run DFS traversal on active nodes from `ActiveNodes` list to find all the active components at the current time step. The process is repeated for each time step in $Time$. Once all nodes in `ActiveNodes` are visited,

we return. The pseudocode in Algorithm 3 provides a high-level description of how the naive implementation works. For simplicity, we add a dummy node to G, which is connected to all the nodes in G and is always active. Then we start the DFS traversal on the dummy node. Thus, in the end, even if G is not a connected graph, we have only a DFS-tree.

Algorithm 3 `Simple_DFS(`G`,`T`,`r`)`

Input: Static undirected graph G, tree T, starting node r
Ensure: Computed DFS-tree T Between Active Nodes rooted at the dummy node and C_t

```
T.adjList.push_back(r)
r.visited ← true
for all ngbr ∈ r.neighbours do
    if ngbr.visited = false and ngbr.active = true then
        add edge (r, ngbr) to T
        remove ngbr from r.neighbours
    end if
    Simple_DFS(G,T,ngbr)
end for
```

Limitation:

The baseline method is inefficient because sometimes it recomputes active components over multiple time steps. For illustration, in Figure 4.4 which shows an example of our problem over three time steps, baseline recomputes the whole DFS at each time step. However, when we look at changes from $t = 1$ to $t = 2$, we know that only node C became active, and other nodes' status did not change. So we only need to check node C neighbours, and if some of them were active, merge those active components and add node C to it. Alternatively, if they were all inactive, make a new active component with node C in it. This overhead of the baseline algorithm is again repeated from $t = 2$ to $t = 3$.

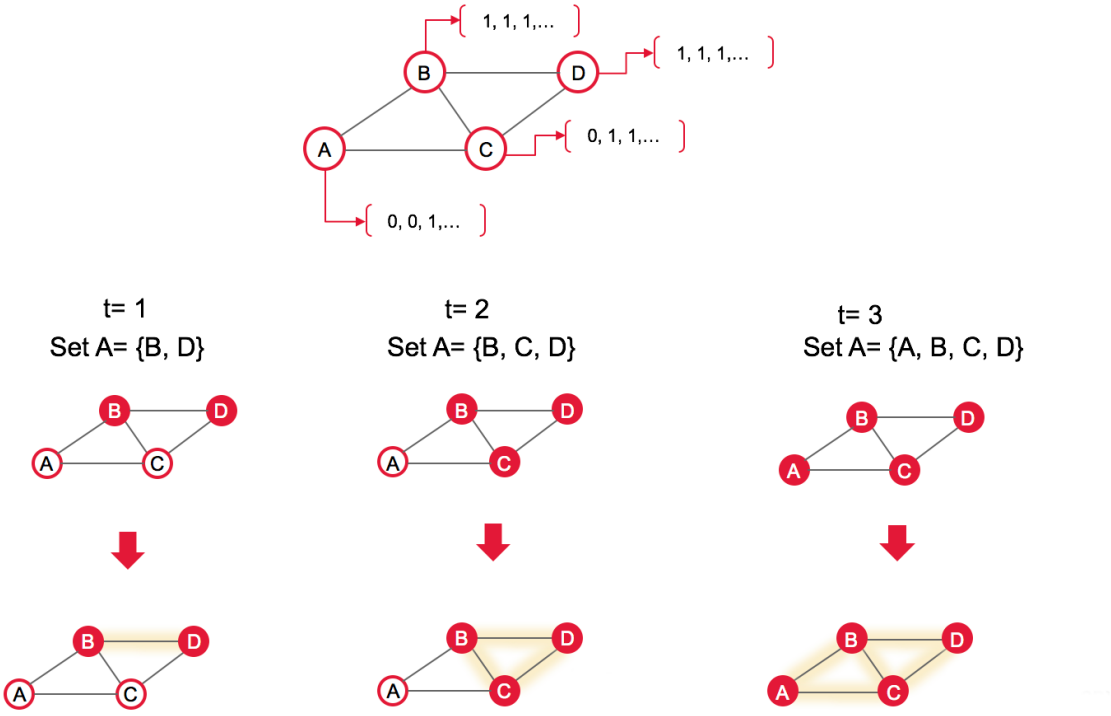


Figure 4.4: Example of the baseline on $t = 1, 2, 3$

4.3 *ActiveComp*: A Faster DFS Algorithm

4.3.1 Overview

The algorithm we use is a modified version of one introduced by Baswana et al. [7]. They address the problem of maintaining a DFS-tree in the subgraph model. However, since we only care about maintaining the active components, it suffices for us to maintain any rooted spanning tree. In particular, we modify their algorithm to instead compute any rooted spanning tree (not necessarily a DFS-tree) of the graph after each set of vertex updates, improving on their algorithm's time complexity by a factor of $\log n$ per time step.

First, we use the preprocessing step of Baswana et al. [7] at time step zero, which is the

fundamental part that will help us build a rooted spanning tree of the active nodes faster than computing it with a DFS algorithm on G at each time step.

In the preprocessing part, we make a DFS-tree T of G and the Shallow Tree ST of T assuming that all vertices are active. We make the data structure (which we explain in 4.3.3) in preprocessing time as well. Then, we add the dummy node to G, which is connected to all the other nodes in G and is permanently active. In this way, at each time step, if we build a spanning tree of active nodes rooted at the dummy node, our active components are the subtrees rooted at each child of the dummy node.

Next, as in Baswana et al. [7], at each time step, using X_t, X_{t-1}, we update the ST and delete inactive nodes from the ST. As an overview of **updating the shallow tree**; we do the following three steps:

1. From each path in ST we delete inactive nodes in it and it will break the path into some pieces. We add each piece as a new `nodePath` to ST and delete the original `nodePath`.

2. Set the `nodePath` for all active nodes.

3. Set the parent of each `nodePath` to the `nodePath` of the first active ancestor of the `nodePath` start node. By first we mean the first one along the route from the start node to the dummy node in other words it will be the nearest active ancestor of the start node.

Time complexity of updating ST is $O(n)$ time with doing all the above steps along a DFS on T, we make sure that the DFS is visiting nodes in order of their dfs_ids.

Then we start to build a rooted spanning tree T^* by running a DFS on the ST. Intuitively by running a DFS on shallow tree ST we mean that we consider each path as a super-node, and when we visit a path through one of its node members, we attach the whole path to the partially grown T^*. Then for each node on that path, we make a list called `Efficient_AL` that is a subset of its adjacency list in the G (in section 4.3.2 we explain how we compute this list) the idea is from [7]. Then we continue the DFS by calling the DFS recursively on the `Efficient_AL` of each node's of the currently added path to T^*.

We call this DFS funtion on a shallow tree `ActiveComp` and the pseudocode of `ActiveComp` is in Algorithm 4. We pass three more auxiliary inputs to this function which can be built from G. Those are T which a DFS-tree of G, T^* which is a tree object (the spanning tree to be) that we make it through the recursive calls, and node x which is the starting node of the recursive DFS algorithm. We assume that matrix X is a global variable that every function can access it and read the nodes' status from it.

Algorithm 4 `ActiveComp`(G, T, T^*, x)

Input: Static undirected graph G, DFS-tree T of G, Spanning-tree T^*, starting node x
Ensure: Computed Spanning-tree of Active Nodes T^* rooted at the dummy node and C_t

```
Attach the nodePath(x) from x to T*
Compute_Efficient_AL(nodePath(x), T)
for all node ∈ nodePath(x) do
  for all u ∈ node.Efficient_AL do
    if u.visited = true then
      continue
    end if
    T*.AdjacencyList.push_back(u)
    T.dfsOrderedList[node.dfs_id].children.push_back(u)
    u.par ← T.dfsOrderedList[node.dfs_id]
    u.children.clear()
    ActiveComp(G, T, T*, u)
  end for
end for
```

4.3.2 Computing Efficient Adjacency List

Based on the earlier discussion about the Shallow Tree section, vertices in a `nodePath` can only have edges to vertices of the `nodePath`'s ancestors or descendants in the shallow tree. Assume we want to fill `Efficient_AL` of `nodePath` p, and we have a function `query`(x, p) that for a vertex x and p returns a vertex on the p that is connected to x if there is any otherwise returns *nullptr*. In this function x has to be one of the p.startNode's descendant that doesn't lie on p. With this function we can make queries for any node on p and fill it's `Efficient_AL` in the two following steps:

1. For each ancestor *anctr* of p we start making queries on vertices of p and *anctr* until we find an edge connecting p and *anctr* together. So as soon as we find an edge connecting *anctr* to p we are done here and will make queries on the next ancestor because that

edge is enough to help us build the spanning tree.

2. For each descendants *dscnt* of p we start making queries on vertices of *dscnt* and p until we find an edge connecting *dscnt* and p together. This step is almost the same as the first step except in the first step the input `nodePath` of function `query` is the ancestor that we are processing but in this step the input `nodePath` is p. This is because of the data structure that we're using and tried to make it smaller in order to less memory consumption. Algorithm 5 shows the details of this fuction.

Algorithm 5 `Compute_Efficient_AL`(T, p)

Input: DFS-tree T of G, `nodePath` p
Ensure: Compute Compute_Efficient_AL for nodes in p

```
mu ← T.dfsOrderedList[p.start.dfs_id].nodePath.par
while node ∈ nodePath(x) do
  for all n ∈ p do
    if mu.end.visited = true then
      break
    end if
    w ← query(T.dfsOrderedList[n.dfs_id], mu)
    if w != nullptr then
      T.dfsOrderedList[n.dfs_id].Efficient_AL.insert(w)
      break
    end if
  end for
  mu ← mu.par
end while
for all id ∈ [p.end.dfs_id + 1, p.start.dfs_id + p.start.sizeOfsubT) do
  u ← T.dfsOrderedList[id]
  if u.visited = true then
    id ← u.nodePath.end.dfs_id
  end if
  if u.active = true and u.visited = false then
    w ← query(u, p)
    if w != nullptr then
      w.Efficient_AL.insert(u)
      id ← u.nodePath.end.dfs_id
    end if
  end if
end for
```

4.3.3 Querying the Data Structure

Using the property of DFS-trees that we described in section 4.1.1 we build a data structure introduced by Baswana et al. [7] of the original graph and use it for answering queries faster. The idea of the data structure is that for each node u of the original graph and its corresponding DFS-tree T we save u's ancestors in T that u has an edge to. We also save

them in increasing order in terms of their `dfs_id`s to help us answer queries faster by doing a binary search. We call this data structure *Anc_Nbr* an array of size n if n is the number of nodes in the original graph. Each cell with index i of this array is an ordered set that saves i'th node's ancestors with an edge to it. For instance, for the graph in Figure 4.2b first we calculate its `dfsOrderedList` as follows; in each tuple the first element is the node id and the second element is the node's `dfs_id`:

$$(1,1),(2,2),(5,3),(6,4),(9,5),(8,6),(7,7),(10,8),(11,9),(4,10),(3,11)$$

Then, we calculate the *Anc_Nbr* data structure for this graph, as shown in Table 4.1. The first row shows node ids, and the second row shows the set of node ids of ancestors that also have a direct edge to the corresponding node. The node ids in each ancestor set are ordered by increasing order of their `dfs_id`s. For example, for node id 7 we save {8, 6}, because these are the ancestors of node 7 that have a direct edge to it, and the order is determined by the fact that node 6's `dfs_id` is 4 and node 8 `dfs_id`.

node id	1	2	3	4	5	6	7	8	9	10	11
ancestors	{}	{1}	{2,4}	{1,2}	{2}	{5}	{6,8}	{6,9}	{6}	{9}	{9,10}

Table 4.1: Data Structure *Anc_Nbr*

The Algorithm 6 shows the detail of computing the *Anc_Nbr* data structure. If we look at this algorithm there are two nested *for* loops the first one is over all the nodes in G and the second one is over nodes' neighbours. The **time complexity** of making this data structure is $m + n$ while n is the number of nodes in G and m is the number of edges of G.

Algorithm 6 `Computing_Anc_Nbr`(G, T)

Input: Static undirected graph G, tree T
Ensure: Computed data structure Anc_Nbr based on T

```
for all u ∈ G.AdjacencyList do
  node_id = u.id
  for all ngbr ∈ u.neighbours do
    if ngbr.dfs_id < u.dfs_id then
      Anc_Nbr[node_id].insert(ngbr)
    end if
  end for
end for
```

Now, for answering a `query`$(node, path)$ all we need to do is to search for the `dfs_id`s of all nodes sitting on the *path* in the set corresponding to the *node* in Anc_Nbr. As this set is ordered, and nodes on the same path have consecutive integer ids, we can accomplish this using a binary search. Algorithm 7 provides the details of the function `query`. Since the maximum size of each set in Anc_Nbr is n and `query` performs a binary search on one of these sets, the worst-case **time complexity** of `query` is $O(\log(n))$.

Algorithm 7 `query`$(node, path)$

Input: node $node$ of G, `nodePath` path
Ensure: Return a pointer to a node on $path$ that is connected to $node$ in G if there is not any return $nullptr$

```
ancestors ← Anc_Nbr[node.dfs_id]
pStart ← path.start
pEnd ← path.end
min ← 0, max ← ancestors.size() − 1
while max > min do
    mean ← (max + min)/2
    if ancestors[mean].dfs_id ≥ pStart.dfs_id and
    ancestors[mean].dfs_id ≤ pEnd.dfs_id then
        return ancestors[mean]
    else if ancestors[mean].dfs_id < pStart.dfs_id then
        min ← mean + 1
    else
        max ← mean − 1
    end if
end while
if min = max then
    if ancestors[mean].dfs_id ≥ pStart.dfs_id and
    ancestors[mean].dfs_id ≤ pEnd.dfs_id then
        return ancestors[mean]
    end if
end if
return nullptr
```

4.4 Time Complexity Analysis

The input of the problem is $G = (V, E)$ where $|V| = n$ and $|E| = m$. In the preprocessing part we make T in $O(m + n)$ time as it's a Depth First Search. Then we build the data structure Anc_Nbr based on T in again $O(m + n)$ time as shown earlier. Updating ST based on inactive nodes as shown earlier can be done in $O(n)$ time. By looking at the Algorithm 4 we see that the time complexity of `ActiveComp` procedure is bounded by the time complexity of `Compute_Efficient_AL` procedure. Also the `Compute_Efficient_AL` procedure is bounded

by the number of calls to the function `query`. For counting the number of calls to function `query` we calculate from each node $u \in V$ how many calls we make. Under the two following scenarios we make a `query` which the first input node is u.

- First one is when w is not visited and one of w.`nodePath`'s ancestors is being attached to the partially grown T^*.
- Second scenario is when w.`nodePath` is being attached to T^* so we make queries from w to all of w.`nodePath`'s ancestors who are not attached yet.

So in total for w and each of w.`nodePath`'s ancestors we make exactly one query. The maximum number of ancestors for a `nodePath` in ST is the height of ST. Assume the height of ST is d so the maximum number of calls to `query` from all of n nodes is nd. Since each query can be done in $O(\log(n))$ time, the time complexity of `Compute_Efficient_AL` procedure is $O(nd\log(n))$. So the time complexity of `ActiveComp` is also $O(nd\log(n))$.

Chapter 5

Experimental Evaluation

The first section of this chapter explains how we generate synthesis data. The second section explains the experimental setup, and at the end, the third section shows the performance of our algorithm compared to three baselines.

For experiments and evaluations we used Python [44] and more specifically NetworkX [20], Matplotlib [28], Numpy [22], Pandas [35], Random [42], Math [43] libraries.

5.1 Synthetic Data Generation

First we explain how to generate different topologies for input graphs and then we explain how we generate matrix X, which determines nodes' status through time.

5.1.1 Graph Generation

We would like to evaluate the performance of the proposed algorithm on graphs of different typologies. We generate different graphs as input of the problem from the Small-World network model [46]. This model generates graphs with lots of triangles, and each node is reachable from any other nodes by a small number of hops, while the number of neighbours

for most of the nodes is very small compared to the size of the network. Specifically, it generates graphs with a not-small clustering coefficient. At the same time, the average distance L between two random nodes is small and grows proportionally to the logarithm of n ($L \propto \log(n)$).

The clustering coefficient for a node v in graph $G = (v, E)$ is noted as C_v and is defined as follows. If N_v is the set of v's neighbours at most, there are $|N_v|(|N_v| - 1)/2$ edges between them and C_v is the proportion of existing ones, in a formal way it is:

$N_v = \{u \in G | (u, v) \in E\}$

$EN_v = \{(i, j) | i, j \in N_v, (i, j) \in E\}$

$C_v = \frac{|EN_v|}{(|N_v||N_v-1|)/2}$

The Clustering Coefficient for a graph C is the average of the Clustering Coefficients of all of its nodes. C_v is a measure that shows what proportion of v's neighbours are also neighbours of each other. Thus C shows how nodes in G tend to form clusters between themselves. A graph with high C has more clusters and is more like social networks [46].

The parameters that we can change in a generated graph that is built with the Small-World network model are as follows:

- n: number of nodes of the graph
- k: number of initial neighbors for each node
- p: edge rewiring probability

Intuitively the algorithm for generating these graphs for the three parameters above is

like this:

1. First, we give all nodes a random ordering and then an id from 1 to n.
2. Then, for each node, we connect it to the next k nodes.
3. Then, for each neighbour of a node with the probability p we change the neighbour to a random another node in the network. We call this step rewiring.

With changes in n we get different sizes for the graph we generate. With changes in k we get different densities for the graph we generate. With changes in p we get different types of graphs. There are three categories:

- When p is less than 0.005 then L and C are both big, and we get a **regular network**.
- When p is bigger than 0.2 the generated graph is like a **random network**.
- When p is between these two values, the generated graph has small L and not small C, which is the definition of a Small-World network. Thus, in this case, we get a **Small-World network**.

For better illustration, we can look at the following plot. In the figure below, we can see how L (mean vertex-vertex distance) and C change with changes in p. We can also see three intervals for p, resulting in the three categories in generated graphs.

5.1.2 Time Series Data Generation

As we mentioned in section 4.2 another input of the problem beside a graph is a n by T_c matrix where n is the number of the nodes of the graph and T_c is the number of time steps.

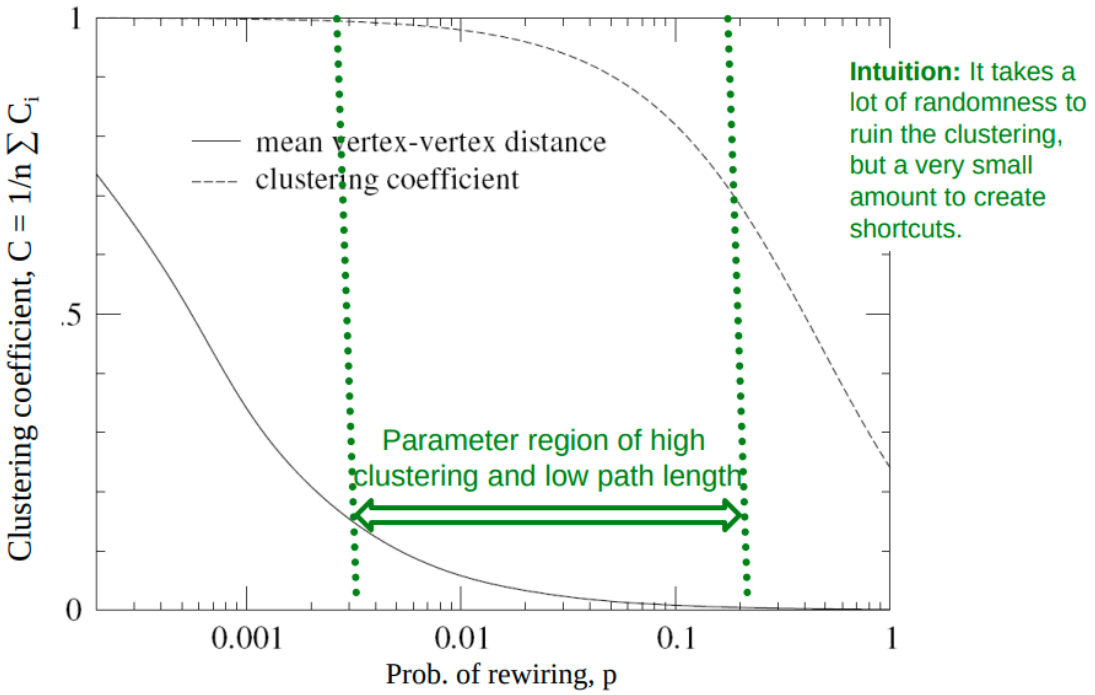


Figure 5.1

This matrix shows each node's status through time. The status of node i at time step j is the ith row and jth column, which is 1 if the node is active at time step j and is 0 if it's inactive.

We imitate two following different scenarios for generating matrix X. In the first scenario, nodes become active or inactive just randomly, and there is no dependency between how nodes become active, and it is a general case. In the second scenario, we wanted to add some dependency between nodes close to each other, so they become active if at least one of their neighbours is already active. Of course, we can use many other dependencies. However, this one is a local dependency that also imitates how the traffic grows in a traffic network or how news spreads in a social network starting from news sources. Thus we wanted to see how our algorithm works when some regional areas of the graph become active.

In the following, we describe how we generate scenarios in more detail. An input parameter called *density* for both scenarios determines the percentage of active nodes. So the number of active nodes is $density \times n$.

1. The first scenario is the **random scenario**. In this scenario, with the input parameter *density*, we determine how many nodes are active and randomly select them, considering a uniform distribution over them and flip their status to active in the output matrix.

2. The second scenario is the **forest fire scenario**. Fire starts from a node and propagates to all of the neighbours of that node and all the neighbours of the node's neighbours and continues the same way. The nodes that are covered by fire are active.
   So in this scenario, we choose the number of initial seed nodes `numOfInitials` which are chosen uniformly at random from all of the nodes in the graph. Then we start fire from each start point and let each fire grow for $\frac{density \times n}{\texttt{numOfInitials}}$ nodes.

5.2 Experimental Setup

In this section, we describe the experiment setups that we use to empirically evaluate the algorithm described in section 4. This includes parameters used to generate different input graphs G and matrices X, the different baselines that we use as comparisons against ours, and the measurements we use to do these comparisons. All of the algorithms in our experiments accept three inputs, a graph G, a matrix X, and a time series index set $Time$, where we previously described how these are generated in Sections 5.1.1 and 5.1.2, respectively.

5.2.1 Input and Parameters

For generating a graph we have three parameters: n, k, p. So in our experiments, we cover different graphs choosing different values for these parameters.

Values that we choose for n are from this set: $\{10000, 50000, 100000\}$. The other numbers represent a minimum and median value. We refer to these as a `small`, `medium`, and `large` graphs respectively. Table 5.1 presents the size of different input graphs.

Size	Number of nodes
`small`	10000
`medium`	50000
`large`	100000

Table 5.1: Different graph sizes

Values that we choose for k are a percentage of n, so for each n we choose a k from this set: $\{0.5\%n, 1\%n, 1.5\%n, 2\%n\}$. As most applications for graphs like social networks or traffic network or internet hubs are sparse graphs [39, 45, 40], we also tried to make graphs with small densities, so we set the maximum to 2% and have an arithmetic sequence with a common difference of 0.5%.

We can also have 3 different values for p from this set $\{0.1, 0.01, 0.5\}$ as it covers all three categories that we described in section 5.1.1. Table 5.2 presents different topologies of input graphs.

To cover the different scenarios of activity for each node, we have two types of input file generated by these scenarios: $\{random, forestfire\}$ which we described in section 5.1.2. Table 5.3 presents different scenarios for generating matrix X.

Topology	Value of p
`regular network`	$p < 0.005$
`small-world network`	$0.005 < p < 0.2$
`random network`	$p > 0.2$

Table 5.2: Different graph topologies

	Scenario
1	random
2	forest fire

Table 5.3: Different scenarios

5.2.2 Evaluation Metric

The focus of our research is on time performance of the various algorithms, therefore we report **the runtime cost** (in milliseconds) for each experiment. Typically, we report the runtime for varying levels of active nodes in the graph, expressed as a percentage of all nodes.

5.2.3 Baselines

We compare the three following algorithms to our algorithm.

1. **Simple DFS [13]**: Simple DFS is the most straightforward known solution for finding the connected components and maintaining the spanning tree in dynamic graphs. But for the cases that we don't have many changes; we don't want to run the DFS algorithm from scratch. So based on how many changes we have between time steps, we want to see when dynamic algorithms perform better than the DFS algorithm.

2. **Dynamic DFS [7]**: This algorithm maintains a DFS-tree, which is also a spanning tree, so we can obtain the list of active components using this algorithm as well. Our

algorithm is based on this Dynamic DFS algorithm with better time complexity by a $\log(n)$ factor. But we maintain a spanning tree.

3. **Edge Deletion [25]**: This algorithm is implemented by Albert et al. [2]. We can use edge deletion algorithms for node deletion by deleting all the incoming and outcoming edges from the node we wish to delete. Unfortunately, we could not compare edge deletion algorithms vs node deletion algorithms on real data, only knowing their time complexity. However, because we had the implementation of this algorithm, we also decided to compare it to our algorithm.

5.3 Results and Discussion

We present the results for graphs with varying topology, size, and percentage of nodes activity. In our experiments, we fix two parameters out of the three (size, topology and scenario) then will change the third parameter to see the effect of that parameter. In the first section, we present the result for when the graph is a Small-World network, and the scenario is random while changing the size of the graph. Then in the second section, we present the result for when the graph is a medium small-world network while the scenario this time is a forest fire. Then in the third section, we show the result for when the graph size is medium, and the scenario is random while changing the topology (regular graph and random graph).

5.3.1 Small-World Network & Random Scenario

In this section, firstly, we talk about the edge deletion algorithm and how it is slower than the other algorithms. Secondly, we show the results under two different groups of applications of

our work. The first group of applications is called **High Percentage Active Nodes** and the second group is called **Low Percentage Active Nodes**. We will describe more detail about these two groups of applications later.

To utilize an edge deletion algorithm for our problem setting when a node x becomes inactive, we have to run the edge deletion algorithm for the number of edges with one endpoint x, making the edge-based algorithms useless for node update settings. This section shows that the edge deletion algorithm performs poorly against the other three algorithms as nodes change in the dynamic graphs in our setting.

Figure 5.2 is a histogram of runtimes for different experiments. All of the experiments are on a graph with $n = 10000$ nodes and $k = 50$. In each experiment, a specific percentage of the nodes are active. Each experiment is 1000 time steps, and in each step, we accept a set of updates that include the active nodes, then we calculate the active components for 1000 times, each of them after each time step. So, for instance, in the experiment where 98% of the nodes are active, we calculate the active components for each of the 1000 time steps based on a set of updates of size $98\%n$, which shows active nodes in that time step.

Figure 5.2 shows how the edge deletion algorithm is slower than the other algorithms. That is why we eliminate this algorithm from future plots to be able to compare other algorithms more clear.

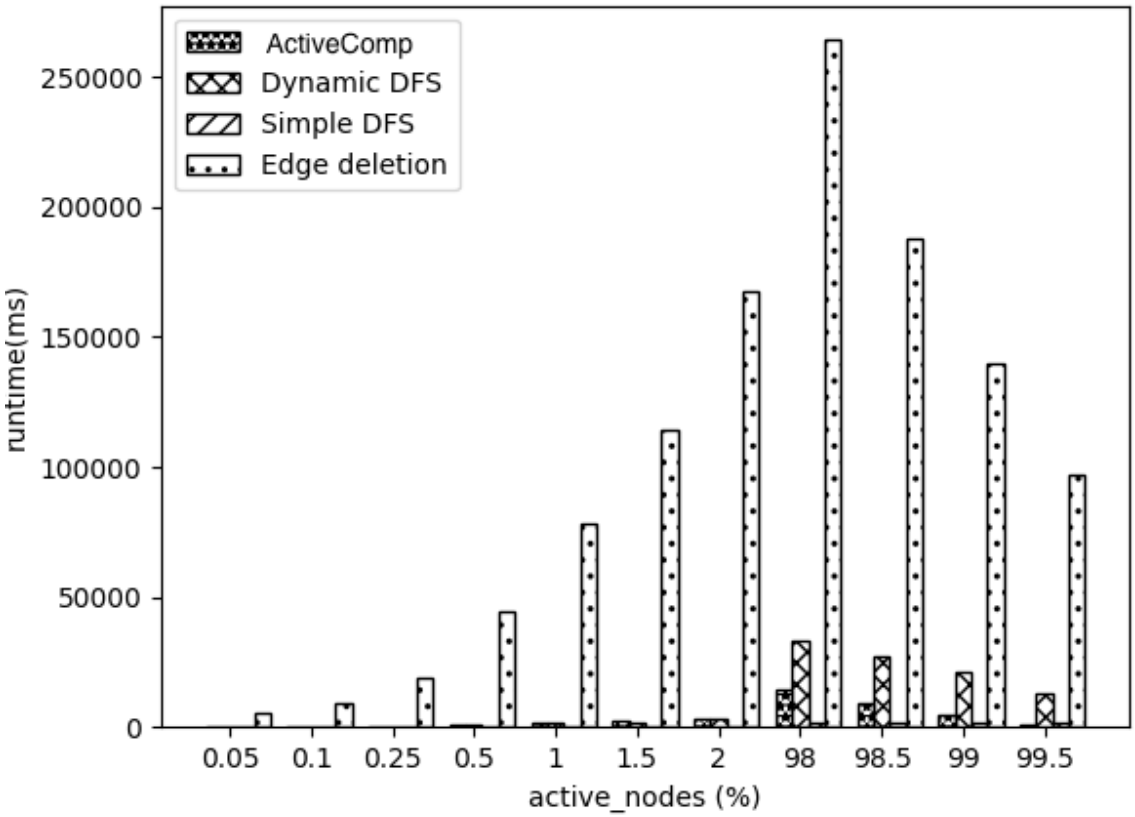


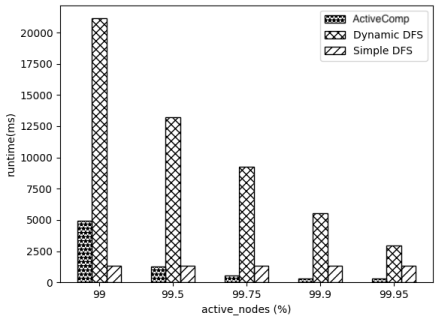
Figure 5.2: Experiment on a Small Graph, Small-World network, k = 50, Random Scenario

High Percentage Active Nodes

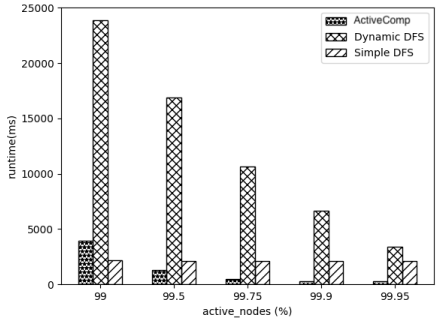
High Percentage Active Nodes are applications where most nodes are active, and only a few become inactive through time. Like networks of hubs or water pipes, when some nodes don't work, we want to see if the network is still connected and find the connected components. In this section, we show how our algorithm performs, in this case, compared to other algorithms.

The following three plots in figures 5.3,5.4,5.5 show the results respectively for $n = 10000$, $n = 50000$, $n = 100000$ and all of the 4 values for k from $0.5\%n$ the top left one to $2\%n$

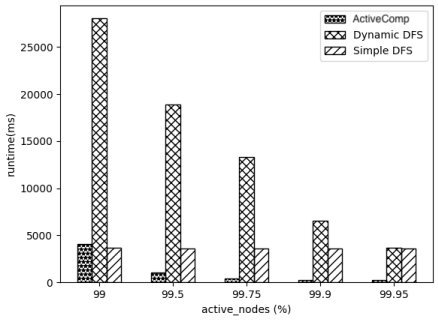
the bottom right one. We show runtime for a few experiments with different input file densities (active nodes percentage) in each plot. For instance, in the figure 5.4a the first experiment is when 99.5% of the nodes with random scenario are active at each time-step. In total, in each experiment, we have 1000 time-steps. This means that we tried to calculate active components 1000 times after each time-step after accepting a set of 0.5% (in this case, 50 nodes) updates that show which nodes are inactive. For each value of k, we run the experiment for different densities of active nodes, and as we can see, the higher the density is, our algorithm performs better than DFS. It shows how we get rid of calculating everything from scratch when only a few nodes aren't active.



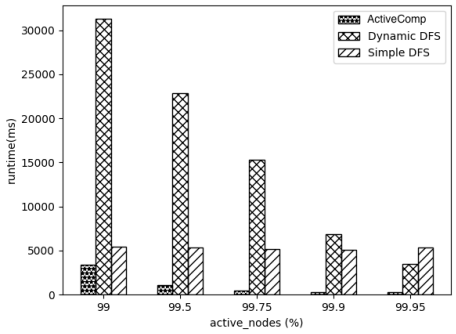
(a) k = 50



(b) k = 100



(c) k = 150

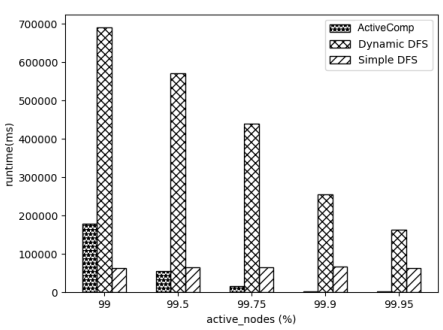


(d) k = 200

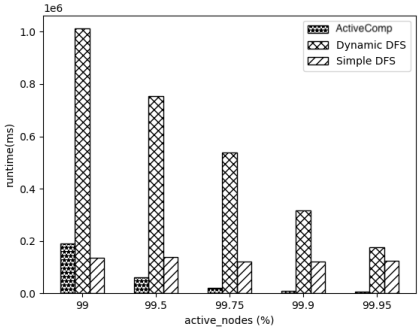
Figure 5.3: Experiments on a Small Graph, Small-World network, Random Scenario

In these plots 5.3 we can see how our algorithm performs better when we have bigger values for k, which make the graph denser. For example, for $k = 50$, our algorithm performs better than the DFS algorithm when at least 99.5% of the nodes are active, while for $k = 200$, our algorithm performs better when at least 90% of the nodes are active. In other words, our algorithm performs better in more cases for $k = 200$ than when $k = 50$. This is because the denser the graph is, the slower the Simple DFS algorithm becomes. In contrast, ActiveComp does not become slower because it utilizes a data structure that becomes bigger when we have a denser graph. This is because time complexity of Simple DFS depends on number of edges but ActiveComp time complexity doesn't depend on the number of edges. In contrast ActiveComp algorithm works better when the graph is denser as we have more information saved.

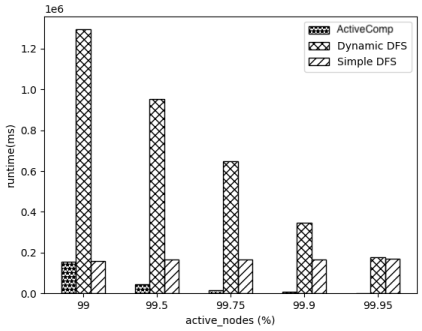
Because most of the applications of finding active components are sparse graphs, we did not go higher than $2\%n$ for k.



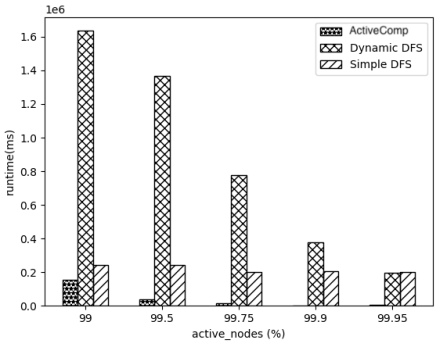
(a) k = 250



(b) k = 500



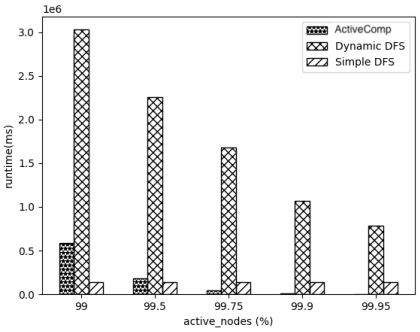
(c) k = 750



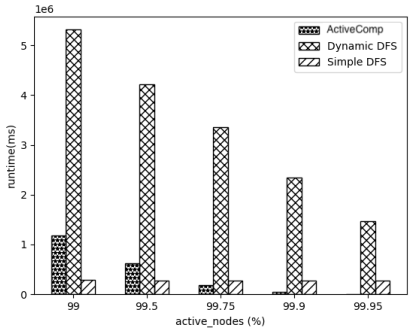
(d) k = 1000

Figure 5.4: Experiments on a Medium Graph, Small-World network, Random Scenario

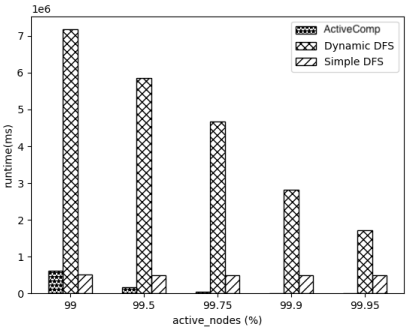
Figure 5.4 shows the same results for a Medium graph. The denser the graph becomes, the slower Simple DFS performs while not affecting ActiveComp considerably. Runtimes became around 40 times bigger while the size of the graph became five times bigger compared to previous experiments, figure 5.3 when the size of the graph was small.



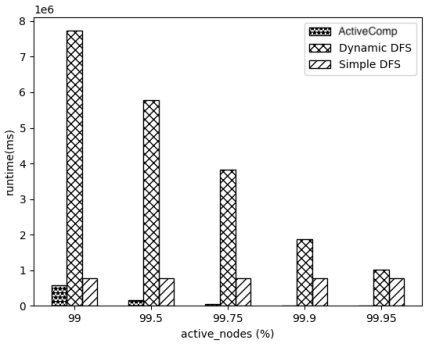
(a) k = 500



(b) k = 1000



(c) k = 1500



(d) k = 2000

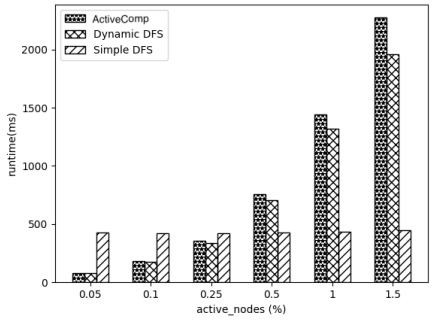
Figure 5.5: Experiments on a Large Graph, Small-World network, Random Scenario

Figure 5.5 shows the same results for a Large graph. Except runtimes became around three times bigger than the previous experiment, 5.4 on a medium graph while the size of the graph became five times bigger.

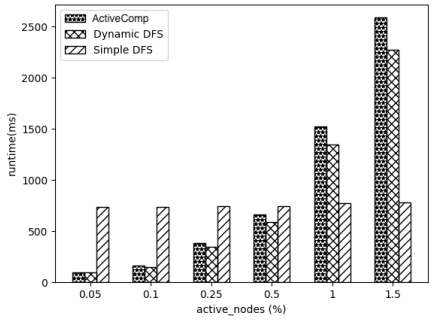
Low Percentage Active Nodes

Other applications of our problem are when only a few nodes of the graph are active, and we want to find the active components without recomputing everything from scratch. Like traffic networks, when intersections are nodes and roads are the edges of the graph, and when an intersection has heavy traffic, we consider it as active, and we want to see what area of the map has heavy traffic. In this section, we show how our algorithm performs, in this case, compared to other algorithms.

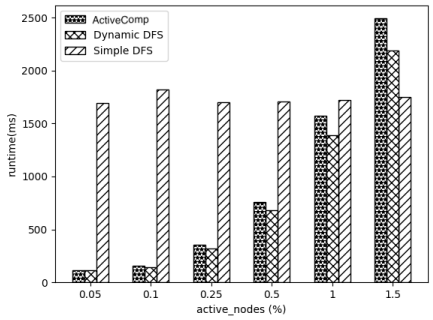
The following three plots in figures 5.6,5.7,5.8 show the results respectively for $n = 10000$, $n = 50000$, $n = 100000$ and all of the 4 values for k from $0.5\%n$ the top left one to $2\%n$ the bottom right one. We show runtime for a few experiments with different inputfile densities (active nodes percentage) in each plot. For instance, in the figure 5.6a the first experiment from left is when 0.25% of the nodes (25 nodes) with random scenario are active at each time-step. In total, in each experiment, we have 1000 time-steps. This means that we tried to calculate active components 1000 times after each time-step after accepting a set of 0.25% (25 nodes in this case) updates that shows which nodes are active. For each value of k, we run the experiment for different densities of active nodes, and as we can see, the lower the density is, the better our algorithm performs compared to DFS. It also is comparable with the Dynamic DFS algorithm. In low percentage active nodes, the running time for all algorithms is generally low. Our algorithm is also a few milliseconds slower than Dynamic DFS in some experiments, which is not noticeable.



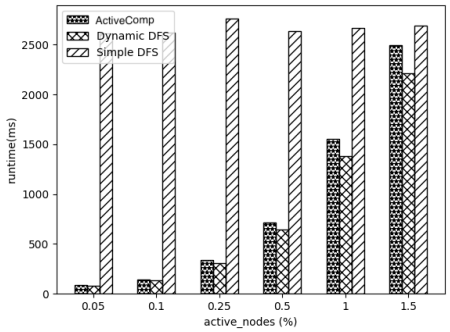
(a) k = 50



(b) k = 100

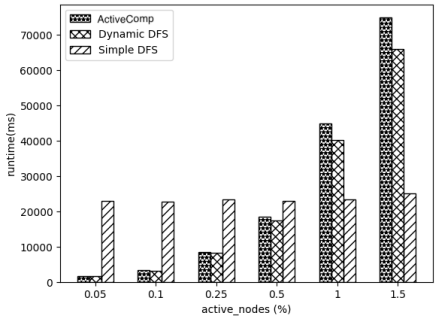


(c) k = 150

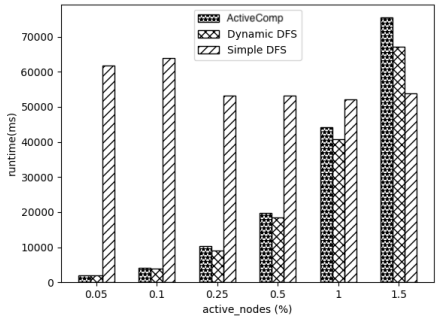


(d) k = 200

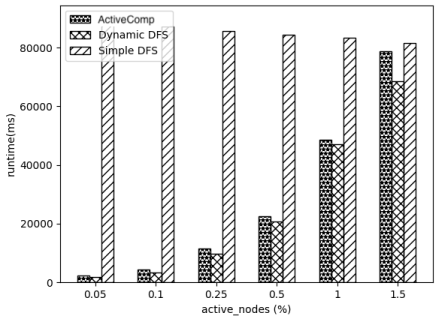
Figure 5.6: Experiments on a Small Graph, Small-World network, Random Scenario



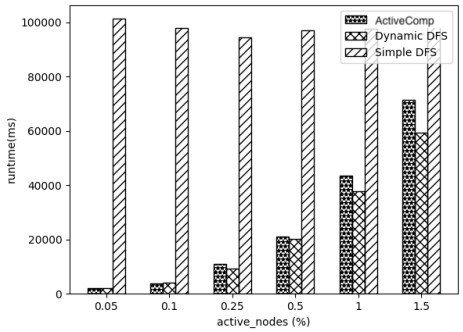
(a) k = 250



(b) k = 500

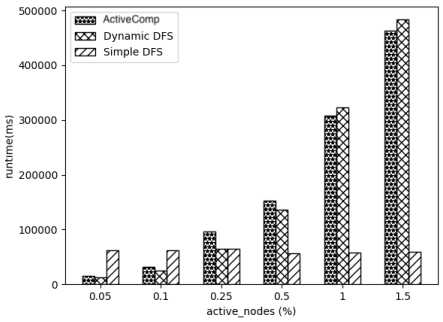


(c) k = 750

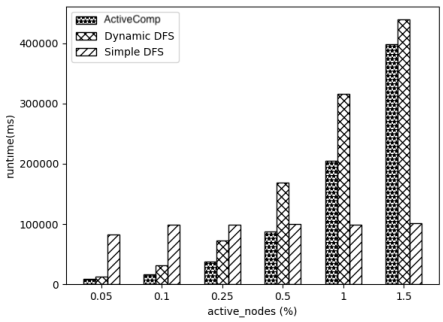


(d) k = 1000

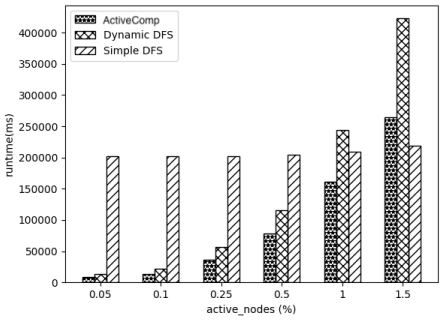
Figure 5.7: Experiments on a Medium Graph, Small-World network, Random Scenario



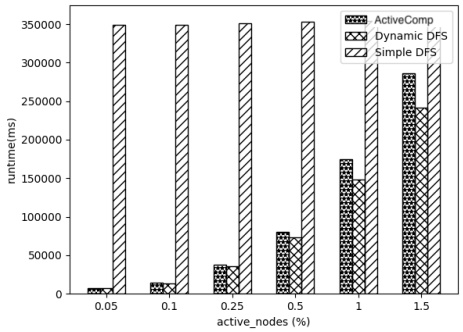
(a) k = 500



(b) k = 1000



(c) k = 1500



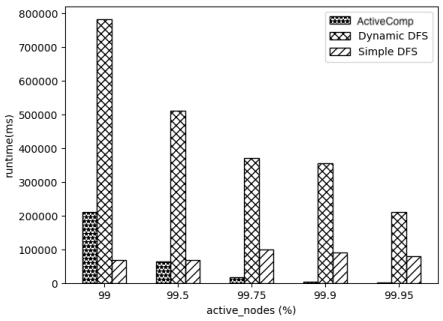
(d) k = 2000

Figure 5.8: Experiments on a Large Graph, Small-World network, Random Scenario

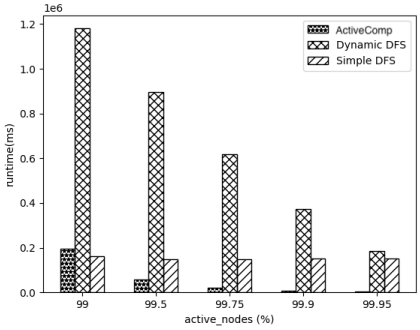
Considering the results of the High Percentage Active Nodes and Low Percentage Active Nodes sections together demonstrates that our algorithm is superior to other algorithms, since it is faster in High Percentage Active Nodes applications and comparable in Active Nodes and Low Percentage Active Nodes.

5.3.2 Small-World Network & Forest fire Scenario

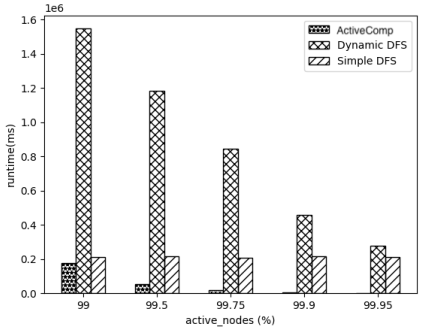
For this scenario, we fix other parameters as we want to see the effect of the forest fire Scenario. We use a medium graph with $n = 50000$, $p = 0.1$ to make it a Small-World network, which makes it a Small-World Network and the same set for $k = \{250, 500, 750, 1000\}$. The result has not changed compared to the Random Scenario in either the High or Low Percentage Active Nodes as we can see in the two following Figures 5.9, 5.10.



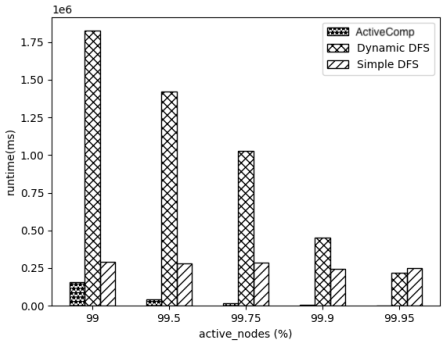
(a) k = 250



(b) k = 500

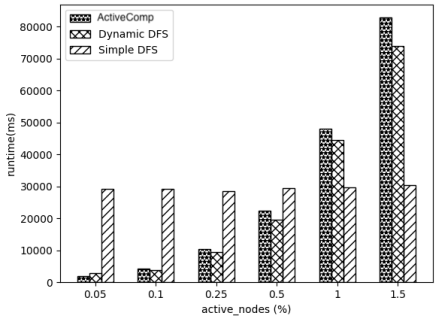


(c) k = 750

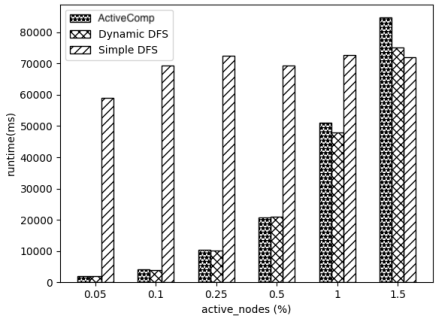


(d) k = 1000

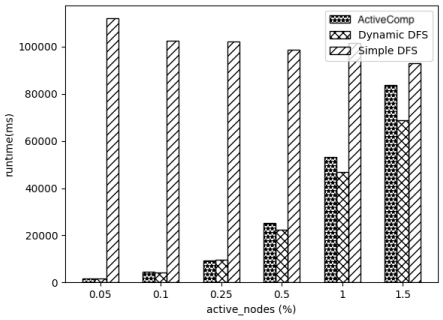
Figure 5.9: Experiments on a Medium Graph, Small-World network, Forest fire Scenario



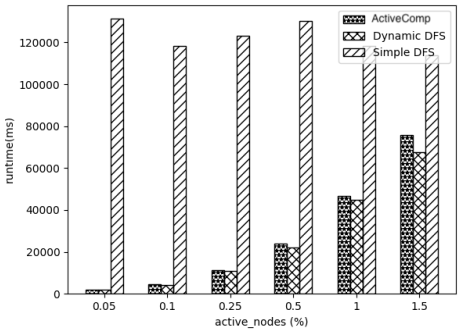
(a) k = 250



(b) k = 500



(c) k = 750



(d) k = 1000

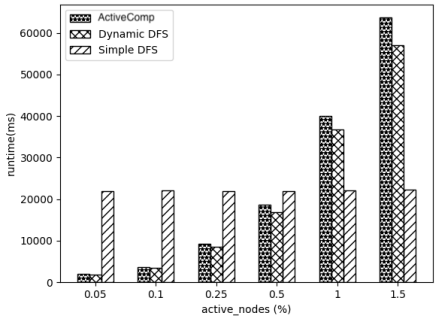
Figure 5.10: Experiments on a Medium Graph, Small-World network, Forest fire Scenario

5.3.3 Other input graphs & Random Scenario

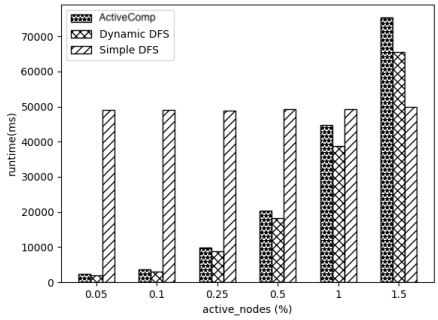
We also run the experiments on different types of graphs, which we introduced in section 5.1.1 as random graphs and regular graphs.

Regular Graphs

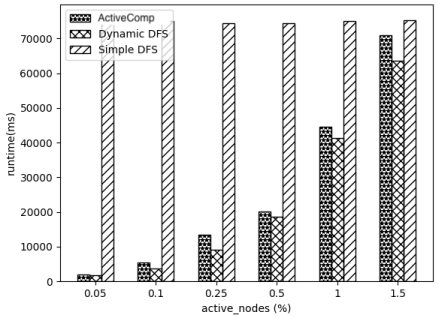
The following results are for regular graphs: the graph's size is medium, and the scenario is random. For low percentage active nodes experiments in Figure 5.11, for $k = 750$ and $k = 1000$, for all values of p our algorithm performs better than simple DFS algorithm. But for all values k and p, the dynamic DFS performs insignificantly faster than our proposed algorithm. In contrast, our algorithm performs faster than the dynamic DFS for all values of k and p in a High Percentage Active Nodes application in Figure 5.12. In Small Word Graphs, our algorithm performs faster than the Simple DFS algorithm for densities higher than $99.5\%n$. Still, in this case (Regular Graphs), our algorithm performs faster only for densities higher than 99.75%.



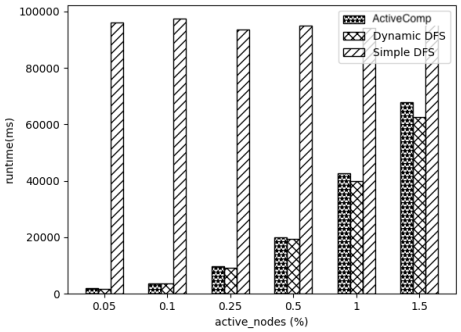
(a) k = 250



(b) k = 500

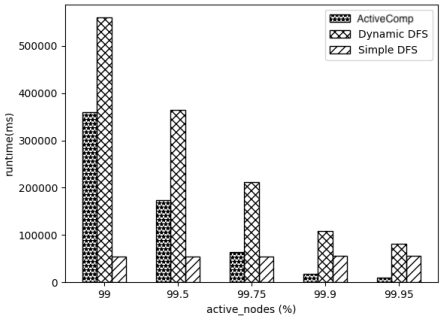


(c) k = 750

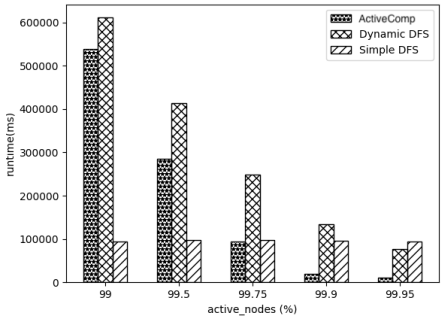


(d) k = 1000

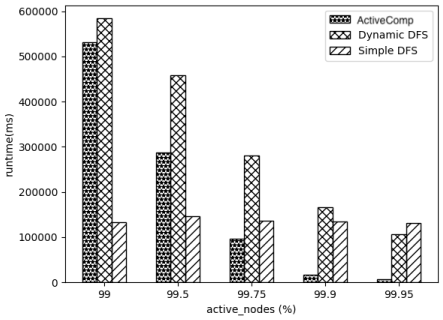
Figure 5.11: Experiments on a Medium Graph, Regular network, Random Scenario



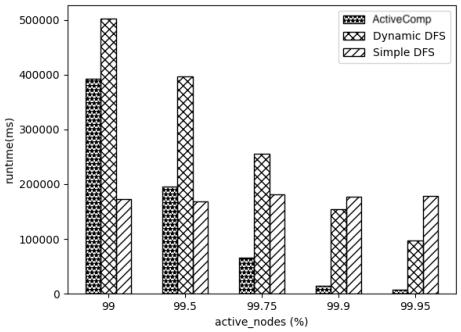
(a) k = 250



(b) k = 500



(c) k = 750

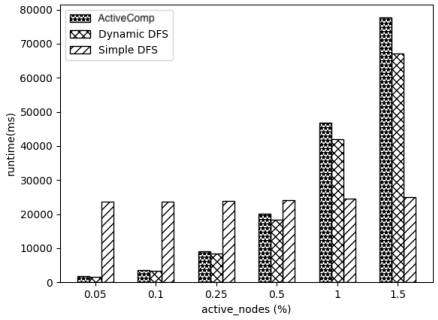


(d) k = 1000

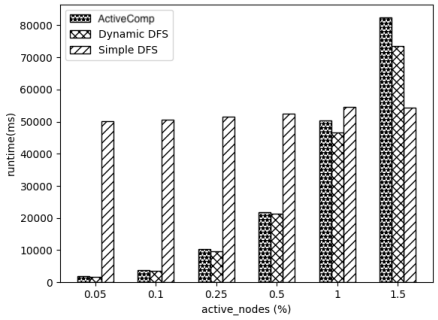
Figure 5.12: Experiments on a Medium Graph, Regular network, Random Scenario

Random Graphs

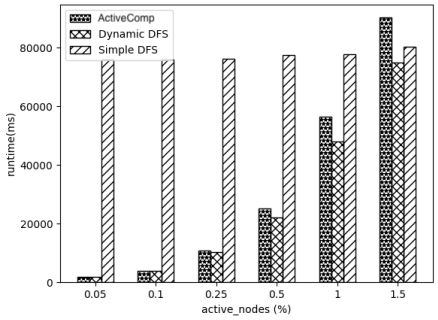
The following results are for random graphs: the graph's size is medium, and the scenario is random. Figure 5.13 is for low percentage active nodes, and the takeaway of the results is the same as the other type of input graphs. But interestingly, in Figure 5.14 our algorithm performs much faster than other algorithms.



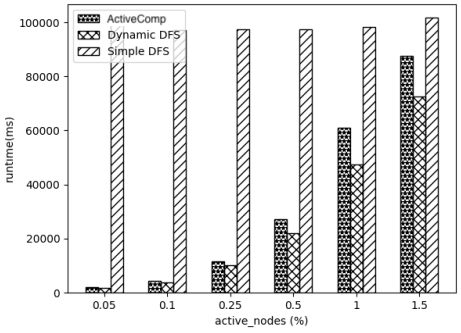
(a) k = 250



(b) k = 500

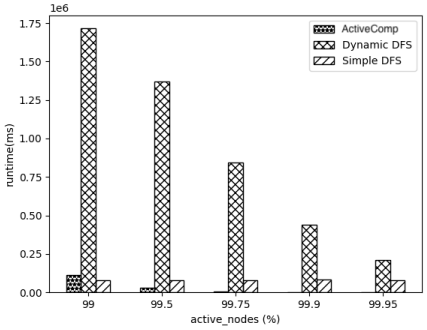


(c) k = 750

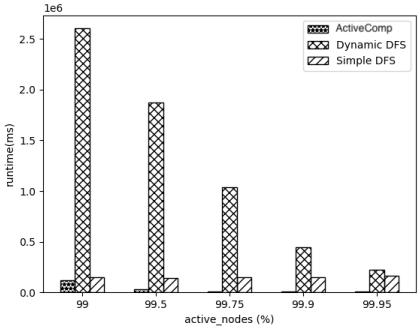


(d) k = 1000

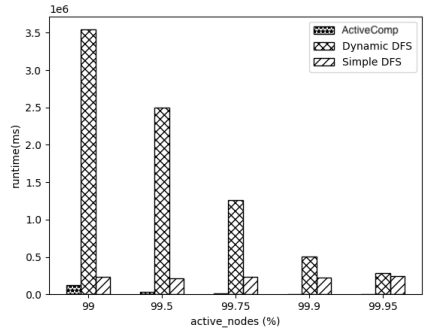
Figure 5.13: Experiments on a Medium Graph, Random network, Random Scenario



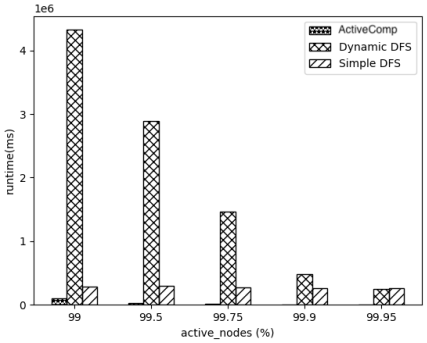
(a) k = 250



(b) k = 500



(c) k = 750



(d) k = 1000

Figure 5.14: Experiments on a Medium Graph, Random network, Random Scenario

Chapter 6

Conclusions and Future Work

6.1 Conclusions

The need to efficiently mine networks of time series can arise in diverse domains and applications. In this research, we focused on scenarios where the network topology is fixed. However, the nodes can become active or inactive over time, while we seek to efficiently compute the set of active components at each time step. This problem can be modelled as a specific type of dynamic graph problem, known as *subgraph model* problems. Instances of the problem can arise in a diverse types of networks, including the transportation network, the internet network, the water distribution network, online social networks, and even the human brain. Monitoring the active components of a network in real-time (while the network is changing due to nodes becoming on/off) is essential for online decision-making. For example, in the transportation network, active components would represent areas in cities where there are groups of interconnected congested intersections (due to traffic). Such information is important for informing alternate routing decisions, deployment of other measures or road design decisions.

This thesis introduced an algorithm for a novel problem called finding active components in a network of time series, where the active components are the connected components of a graph induced to active nodes in a *subgraph model* setting. Our algorithm is a modified version of a work by Baswana et al. [7]. They maintain a DFS-tree in a dynamic graph under vertex update. We improved the time complexity of their algorithm by a log factor. We use their data structure, which saves some of the original graph G edges that are not in the DFS-tree of G. Then we construct a shallow tree based on the DFS-tree. The shallow tree is the DFS-tree when its nodes are partitioned into super-nodes. Then we update the shallow tree based on the nodes that became inactive. Finally, the new spanning tree is maintained by applying a DFS on the super-nodes using a data structure that recovers the lost connections caused by deleting inactive nodes from the DFS-tree.

Previous related work has focused on answering connectivity queries in the subgraph model. However, we maintain and report the whole spanning tree in this research. Technically, this means that we can answer the connectivity queries in $O(1)$ time, while the other works either take more time to answer the connectivity queries or take more time to reconstruct the spanning tree. In addition, some of the parameters introduced in the time complexity analysis of previous works make it challenging to compare their actual performance on real data and under different conditions. We, therefore, perform a comprehensive empirical study including implementations of our method (*ActiveComp*), Baswana's [7] method, as well as two other sensible baseline methods and compare their runtime performance for various settings.

6.2 Limitations

Our method inherits some of the limitations of similar approaches for dynamic graphs. Specifically, it might not scale to very large graphs due to the need to maintain a specialized data structure in memory, and it might be underperforming for specific instances of the problem. We discuss these issues below in more detail.

6.2.1 Scalability to Very Large Graphs

Our method relies on storing an auxiliary data structure in memory during the preprocessing phase. Since this data structure grows with the number of edges in the original graph, the larger and the denser the graph is, the larger the memory requirement is. As such, our method cannot scale to very large graphs. Some ideas on how to scale to larger graphs are presented in Section 6.3.

6.2.2 Underperforming in Specific Instances of the Problem

By design of our method, and as demonstrated in the empirical evaluation, if the percentage of active nodes at a specific time step ranges between 10% and 90% of all nodes in the graph, then the simple DFS algorithm can outperform our algorithm. While these cases are not typical, since most of the changes in the network occur gradually, they could render the algorithm less useful for certain instances of the problem. However, our method would still provide the correct set of active components despite the delay.

6.3 Future Work

There are several directions for future work. Here we briefly discuss some of these ideas.

6.3.1 Employing Parallel Computations

It is possible to design a parallel algorithm (that leverages multiple CPUs) to track the changes in a dynamic graph and find spanning trees over time. This might as well reduce the size of the memory required to store the primary data structure constructed during the preprocessing phase.

6.3.2 Fine-tuning to Accommodate Different Network Topologies

The method we presented would perform differently depending on the topology of the input graph. One could consider employing slightly different data structures to accommodate the special type of networks, such as tree-like graphs or dense graphs. One could therefore tune the algorithm to the topology of the input graph with the objective of making it faster.

6.3.3 Extending the Comparative Empirical Analysis

As mentioned earlier, there is a gap between theoretical studies for dynamic graphs and empirical studies. While the theoretical results of a method's time complexity guarantee its performance regarding the worst (or average) case, it is unclear how the method would actually perform on real data. Future work could implement related algorithms from the literature and compare their performance against our method for different network topologies and varying parameter values.

6.3.4 Employing Real Data and Scenarios

To obtain better insights into the different methods' comparative performance, one can employ real data and scenarios instead of synthetic data. Real scenarios can exhibit special cases that are not easily captured by controlled parameters of the synthetic data, leading to more bottlenecks or providing further opportunities and ideas for better method design.

Bibliography

[1] Abbe, E., Bandeira, A. S., and Hall, G. Exact recovery in the stochastic block model. *IEEE Transactions on Information Theory 62*, 1 (2015), 471–487.

[2] Alberts, D., Cattaneo, G., and Italiano, G. F. An empirical study of dynamic graph algorithms. *Journal of Experimental Algorithmics (JEA) 2* (1997), 5–es.

[3] Anagnostopoulos, A., Łącki, J., Lattanzi, S., Leonardi, S., and Mahdian, M. Community detection on evolving graphs. In *Advances in Neural Information Processing Systems* (2016), pp. 3522–3530.

[4] Baswana, S., Chaudhury, S. R., Choudhary, K., and Khan, S. Dynamic dfs in undirected graphs: breaking the o (m) barrier. In *Proceedings of the twenty-seventh Annual ACM-SIAM Symposium on Discrete Algorithms* (2016), SIAM, pp. 730–739.

[5] Baswana, S., Chaudhury, S. R., Choudhary, K., and Khan, S. Dynamic dfs in undirected graphs: Breaking the o(m) barrier. *SIAM Journal on Computing 48*, 4 (2019), 1335–1363.

[6] Baswana, S., Choudhary, K., and Roditty, L. An efficient strongly connected components algorithm in the fault tolerant model. *Algorithmica 81*, 3 (2019), 967–985.

[7] Baswana, S., Gupta, S. K., and Tulsyan, A. Fault tolerant and fully dynamic dfs in undirected graphs: simple yet efficient. *arXiv preprint arXiv:1810.01726* (2018).

[8] Bhandari, S., Bergmann, N., Jurdak, R., and Kusy, B. Time series data analysis of wireless sensor network measurements of temperature. *Sensors 17*, 6 (2017), 1221.

[9] Borradaile, G., Pettie, S., and Wulff-Nilsen, C. Connectivity oracles for planar graphs. In *Scandinavian Workshop on Algorithm Theory* (2012), Springer, pp. 316–327.

[10] Cai, Y., Tong, H., Fan, W., and Ji, P. Fast Mining of a Network of Coevolving Time Series. In *Proceedings of the 2015 SIAM International Conference on Data Mining* (June 2015), Society for Industrial and Applied Mathematics, pp. 298–306.

[11] Chan, T. M., P a ˘ traşcu, M., and Roditty, L. Dynamic connectivity: Connecting to networks and geometry. *SIAM Journal on Computing 40*, 2 (2011), 333–349.

[12] Chen, L., Duan, R., Wang, R., and Zhang, H. Improved algorithms for maintaining dfs tree in undirected graphs. *CoRR, abs/1607.04913* (2016).

[13] Cormen, T. H., Leiserson, C. E., Rivest, R. L., and Stein, C. *Introduction to Algorithms*, 2nd ed. The MIT Press, 2001.

[14] Cranor, C., Johnson, T., Spataschek, O., and Shkapenyuk, V. Gigascope: a stream database for network applications. In *Proceedings of the 2003 ACM SIGMOD international conference on Management of data* (2003), pp. 647–651.

[15] Duan, R. New data structures for subgraph connectivity. In *International Colloquium on Automata, Languages, and Programming* (2010), Springer, pp. 201–212.

[16] Duan, R., and Pettie, S. Connectivity oracles for failure prone graphs. In *Proceedings of the forty-second ACM symposium on Theory of computing* (2010), pp. 465–474.

[17] Duan, R., and Pettie, S. Connectivity oracles for graphs subject to vertex failures. In *Proceedings of the Twenty-Eighth Annual ACM-SIAM Symposium on Discrete Algorithms* (2017), SIAM, pp. 490–509.

[18] Duan, R., and Pettie, S. Connectivity oracles for graphs subject to vertex failures. *SIAM Journal on Computing 49*, 6 (2020), 1363–1396.

[19] Frigioni, D., and Italiano, G. F. Dynamically switching vertices in planar graphs. *Algorithmica 28*, 1 (2000), 76–103.

[20] Hagberg, A., Swart, P., and S Chult, D. Exploring network structure, dynamics, and function using networkx. Tech. rep., Los Alamos National Lab.(LANL), Los Alamos, NM (United States), 2008.

[21] Hanauer, K., Henzinger, M., and Schulz, C. Recent advances in fully dynamic graph algorithms. *arXiv preprint arXiv:2102.11169* (2021).

[22] Harris, C. R., Millman, K. J., van der Walt, S. J., Gommers, R., Virtanen, P., Cournapeau, D., Wieser, E., Taylor, J., Berg, S., Smith, N. J., Kern, R., Picus, M., Hoyer, S., van Kerkwijk, M. H., Brett, M., Haldane, A., del Río, J. F., Wiebe, M., Peterson, P., Gérard-Marchant, P., Sheppard, K., Reddy, T., Weckesser, W., Abbasi, H., Gohlke, C., and Oliphant, T. E. Array programming with NumPy. *Nature 585*, 7825 (Sept. 2020), 357–362.

[23] Hart, M. G., Ypma, R. J., Romero-Garcia, R., Price, S. J., and Suckling, J. Graph theory analysis of complex brain networks: new concepts in brain mapping applied to neurosurgery. *Journal of neurosurgery 124*, 6 (2016), 1665–1678.

[24] Henzinger, M., and Neumann, S. Incremental and fully dynamic subgraph connectivity for emergency planning. *arXiv preprint arXiv:1611.05248* (2016).

[25] Henzinger, M. R., and King, V. Randomized fully dynamic graph algorithms with polylogarithmic time per operation. *Journal of the ACM (JACM) 46*, 4 (1999), 502–516.

[26] Holm, J., De Lichtenberg, K., and Thorup, M. Poly-logarithmic deterministic fully-dynamic algorithms for connectivity, minimum spanning tree, 2-edge, and biconnectivity. *Journal of the ACM (JACM) 48*, 4 (2001), 723–760.

[27] Huang, S.-E., Huang, D., Kopelowitz, T., and Pettie, S. 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* (2017), SIAM, pp. 510–520.

[28] Hunter, J. D. Matplotlib: A 2d graphics environment. *Computing in Science & Engineering 9*, 3 (2007), 90–95.

[29] Khan, S., and Mehta, S. K. Depth first search in the semi-streaming model. *arXiv preprint arXiv:1901.03689* (2019).

[30] Li, L. Fast algorithms for mining co-evolving time series.

[31] Marshall, S., Gil, J., Kropf, K., Tomko, M., and Figueiredo, L. Street network studies: from networks to models and their representations. *Networks and Spatial Economics 18*, 3 (2018), 735–749.

[32] McGregor, A. Graph stream algorithms: a survey. *ACM SIGMOD Record 43*, 1 (2014), 9–20.

[33] Nakamura, K. Fully dynamic connectivity oracles under general vertex updates. In *28th International Symposium on Algorithms and Computation (ISAAC 2017)* (2017), Schloss Dagstuhl-Leibniz-Zentrum fuer Informatik.

[34] Nakamura, K., and Sadakane, K. A space-efficient algorithm for the dynamic dfs problem in undirected graphs. In *International Workshop on Algorithms and Computation* (2017), Springer, pp. 295–307.

[35] pandas development team, T. pandas-dev/pandas: Pandas, Feb. 2020.

[36] Papadimitriou, S., Sun, J., and Faloutsos, C. Streaming pattern discovery in multiple time-series.

[37] Patrascu, M., and Thorup, M. Planning for fast connectivity updates. In *48th Annual IEEE Symposium on Foundations of Computer Science (FOCS'07)* (2007), IEEE, pp. 263–271.

[38] Rannou, L., Magnien, C., and Latapy, M. Strongly connected components in stream graphs: Computation and experimentations. In *International Conference on Complex Networks and Their Applications* (2020), Springer, pp. 568–580.

[39] Ravazzi, C., Tempo, R., and Dabbene, F. Learning influence structure in sparse social networks. *IEEE Transactions on Control of Network Systems 5*, 4 (2018), 1976–1986.

[40] Sampaio Filho, C. I., Moreira, A. A., Andrade, R. F., Herrmann, H. J., and Andrade, J. S. Mandala networks: ultra-small-world and highly sparse graphs. *Scientific reports 5*, 1 (2015), 1–6.

[41] Sleator, D. D., and Tarjan, R. E. A data structure for dynamic trees. *Journal of computer and system sciences 26*, 3 (1983), 362–391.

[42] Van Rossum, G. *The Python Library Reference, release 3.8.2*. Python Software Foundation, 2020.

[43] Van Rossum, G. *The Python Library Reference, release 3.8.2*. Python Software Foundation, 2020.

[44] Van Rossum, G., and Drake Jr, F. L. *Python reference manual*. Centrum voor Wiskunde en Informatica Amsterdam, 1995.

[45] Wagner, D., and Willhalm, T. Geometric speed-up techniques for finding shortest paths in large sparse graphs. In *European Symposium on Algorithms* (2003), Springer, pp. 776–787.

[46] Watts, D. J., and Strogatz, S. H. Collective dynamics of 'small-world'networks. *nature 393*, 6684 (1998), 440–442.

[47] Yang, B., Wen, D., Qin, L., Zhang, Y., Wang, X., and Lin, X. Fully dynamic depth-first search in directed graphs. *Proceedings of the VLDB Endowment 13*, 2 (2019), 142–154.

[48] Yang, Y., Yu, J. X., Gao, H., Pei, J., and Li, J. Mining most frequently changing component in evolving graphs. *World Wide Web 17*, 3 (2014), 351–376.

[49] Yao, Y., Gehrke, J., et al. Query processing in sensor networks. In *Cidr* (2003), pp. 233–244.

[50] Zhu, Y., and Shasha, D. Statstream: Statistical monitoring of thousands of data streams in real time. In *VLDB'02: Proceedings of the 28th International Conference on Very Large Databases* (2002), Elsevier, pp. 358–369.