Streaming Algorithms and Parameterized Streaming

Natti Bhuvana Sai

A Thesis Submitted to
Indian Institute of Technology Hyderabad
In Partial Fulfillment of the Requirements for
The Degree of Master of Technology

Department of Computer Science and Engineering

June 2015

Declaration

I declare that this written submission represents my ideas in my own words, and where ideas or words of others have been included, I have adequately cited and referenced the original sources. I also declare that I have adhered to all principles of academic honesty and integrity and have not misrepresented or fabricated or falsified any idea/data/fact/source in my submission. I understand that any violation of the above will be a cause for disciplinary action by the Institute and can also evoke penal action from the sources that have thus not been properly cited, or from whom proper permission has not been taken when needed.

Bhuvana

(Signature)

(Natti Bhuvana Sai)

CS13M1006

(Roll No.)

Approval Sheet

This Thesis entitled Streaming Algorithms and Parameterized Streaming by Natti Bhuvana Sai is approved for the degree of Master of Technology from IIT Hyderabad.

M.V. Pandurangarao

(M.V. Panduranga Rao) Examiner
Dept. of COMPUTER SCIENCE Engineering
IITH

N.R. Aravind

N.R. ARAVIND. (Internal) Examiner
Dept. of Computer Science Engineering
IITH

(Dr. Subrahmanyam Kalyanasundaram) Adviser
Dept. of Computer Science and Engineering
IITH

U. Ramakrishna

(U. RAMAKRISHNA) Chairman
Dept. of COMPUTER SCIENCE Engineering
IITH

Acknowledgements

I would like to express my deepest gratitude to my thesis adviser Dr. Subrahmanyam Kalyanasundaram for his guidance and support. His encouragement, creativity and excellent knowledge have always been a constant source of motivation for me. Thanks to God Almighty for the completion of this master's thesis. Last but not the least I am grateful to my parents and my brother for their love, blessings and support throughout this endeavour.

Abstract

Over the last few years, there has been considerable amount of study and work on developing algorithms for processing massive graphs in the data stream model. Storing massive graphs in the memory of a single machine is not practical which is what the motivation behind data stream algorithms. To obtain space and time efficient algorithms, we develop streaming/semi-streaming algorithms where it is reasonable to assume that the input graph arrives as a stream of edges. We can process the input in either one or multiple passes and the working memory space is restricted. In this thesis, we first present the algorithms for processing the directed dynamic graphs in the semi-streaming model which is an open area for research. Semi-streaming model is a variant of streaming model(restricted space is $\mathcal{O}(polylog\ n)$) where the space usage is restricted to $\mathcal{O}(n\ polylog\ n)$ where n is the number of vertices in the graph. We also propose a solution to the open problem suggested by Andrew McGregor in Matching Open Problem. The problem states that "Consider an unweighted graph on n nodes defined by a stream of edge insertions and deletions. Is it possible to approximate the size of the maximum cardinality matching up to constant factor given a single pass and $o(n^2)$ space?".

We present new solutions for finding kernels of the parameterized versions of few graph problems in streaming/semi-streaming models. For each problem, we are provided an undirected graph G and parameter k as input and our goal is to decide whether there is a solution bounded by k. We mainly consider two models for the graph stream in this thesis. First one is the insertion-only model where only edge insertions are possible. The other one is dynamic model where both edge insertions and deletions are possible.

We show the following results :

1. In insert-only model, we present an algorithm for finding linear kernel for edge dominating set where the parameter k is the size of the solution.
2. In insert-delete model, we combine the kernelization technique with sketch structures to present an algorithm for finding linear kernel for edge dominating set where the parameter k is the size of the solution.
3. In insert-only model, we present the first algorithms for finding the linear kernels of undirected feedback vertex set where the parameter k is the size of the solution.

Contents

Declaration . . . ii
Approval Sheet . . . iii
Acknowledgements . . . iv
Abstract . . . v

1 Introduction **1**
1.1 Massive Graphs . . . 1
1.2 Related Work . . . 2
1.3 Thesis Outline . . . 2

2 Data Stream Algorithms **3**
2.1 Data Stream . . . 3
2.2 Data Stream Model . . . 3
2.3 Techniques used . . . 4
2.4 Example . . . 4

3 Streaming Algorithms **5**
3.1 Models in Steaming Algorithms . . . 5
3.2 Graph Streams . . . 5
3.2.1 Variants in Graph Stream Model . . . 6
3.3 Example Algorithms . . . 6

4 New Solutions in Semi-Streaming Model **8**
4.1 Simple Directed Graphs . . . 8
4.1.1 Preliminaries . . . 8
4.1.2 Algorithms for Simple Directed Graphs . . . 9
4.1.2.1 Connectivity . . . 9
4.1.2.1.1 Sketch-Based Algorithm . . . 11
4.2 Matching in Turnstile Streams . . . 15
4.2.1 Definitions . . . 15
4.2.2 Maximal matching in dynamic graph streams . . . 16
4.2.2.1 Solution 1 . . . 16
4.2.2.1.1 Example : . . . 17
4.2.2.2 Solution 2 . . . 19
4.2.2.2.1 Example : . . . 21

5 Parameterized Complexity Concepts **27**
5.1 Parameterized Complexity . . . 27
5.2 Fixed Parameter Tractability . . . 27

5.2.1 Techniques used . . . 28
5.2.1.1 Bounded Search Trees . . . 28
5.2.1.2 Kernelization . . . 28
5.3 Parameterized Streaming . . . 29

6 New Solutions in Parameterized Streaming **30**
6.1 Preliminaries . . . 30
6.2 Edge Dominating Set . . . 30
6.2.1 Definition . . . 30
6.2.2 Existing Work . . . 30
6.2.2.1 Single Pass Kernel . . . 31
6.2.2.2 2 - Pass Kernel . . . 31
6.2.3 Proposed Work . . . 31
6.2.3.1 Multi Pass Kernel for Insert Only Stream . . . 33
6.2.3.2 Multi Pass Kernel for Insert Delete Stream . . . 34
6.3 Undirected Feedback Vertex Set . . . 35
6.3.1 Definition . . . 35
6.3.2 Solution 1 . . . 35
6.3.3 Fixed Parameter Tractable algorithm for Feedback Vertex Set . . . 38
6.3.4 Solution 2 . . . 39

7 Conclusion and Future Work **40**
References . . . 40

List of Figures

4.1.1 Turnstile Stream 8

4.1.2 Turnstile Stream Contd 9

4.1.3 Spanning Forest Example 10

4.1.4 Spanning Forest : Example 1 12

4.2.1 Matching 15

List of Tables

4.1.1 Sketch Table of connectivity : Step 1 12
4.1.2 Sketch Table of connectivity : Step 2 12
4.1.3 Sketch Table of connectivity : Step 2 - Construction of X 12
4.1.4 Sketch Table of connectivity : Final Step 2 13
4.1.5 Sketch Table of connectivity : Step 3 13
4.1.6 Sketch Table of connectivity : Step 3 - Construction of X 13
4.1.7 Sketch Table of connectivity : Final Step 3 13
4.1.8 Sketch Table of connectivity : Step 4 14
4.1.9 Sketch Table of connectivity : Step 4 - Construction of X 14
4.1.10 Sketch Table of connectivity : Final Step 4 14
4.1.11 Sketch Table of connectivity : Step 5 14
4.1.12 Sketch Table of connectivity : Step 5 - Construction of X 14
4.1.13 Sketch Table of connectivity : Final Step 5 14
4.1.14 Sketch Table of connectivity : Step 6 15
4.1.15 Sketch Table of connectivity : Step 6 - Construction of X 15
4.1.16 Sketch Table of connectivity : Final Step 6 15
4.2.1 Matching Sketches Example : 1 22
4.2.2 Matching Sketches Example : 2 22
4.2.3 Matching Sketches Example : 3 22
4.2.4 Matching Sketches Example : 4 23
4.2.5 Matching Sketches Example : 5 23
4.2.6 Matching Sketches Example : 6 23
4.2.7 Matching Sketches Example : 7 23
4.2.8 Matching Sketches Example : 8 24
4.2.9 Matching Sketches Example : 9 24
4.2.10 Matching Sketches Example : 10 24
4.2.11 Matching Sketches Example : 11 25
4.2.12 Matching Sketches Example : 12 25
4.2.13 Matching Sketches Example : 13 25

List of Algorithms

2.3.1 Sampling Algorithm 4
3.3.1 Connectivity Algorithm 6
4.1.1 Connectivity Algorithm in Directed Acyclic Graphs 11
4.2.1 Matching in Turnstile Stream Algorithm : Solution 1 16
4.2.2 Matching in Turnstile Stream Algorithm : Solution 2 20
6.2.1 Edge Dominating Set Streaming Kernel Algorithm : Insert Only Stream 33
6.2.2 Edge Dominating Set Streaming Kernel Algorithm : Dynamic Streams 34
6.3.1 Undirected Feedback Vertex Set : Solution 1 35
6.3.2 UFVS_FPT_Algorithm 38
6.3.3 Undirected Feedback Vertex Set : Solution 2 39

Chapter 1

Introduction

1.1 Massive Graphs

Graphs are the fundamental data structures for many computational applications. Many types of highly structured data can be represented as graphs. For example : data structures, computer networks, pathing and maps, molecules represent massive data which can be modeled as graphs. In many real world scenarios, massive graphs arise naturally. Massive graphs can be either sparse graphs which have a large number of nodes or dense graphs that have a large number of edges. Massive graphs usually arise in any data-relationship applications. Few examples of massive graphs are

i. Social network graphs in which vertices represent people and edges represent relationships. Eg : Facebook graphs.

ii. Call graphs in which vertices represent users and edges represent the calls made from one user to the other.

iii. Web graphs in which vertices represent web-pages and edges represent hyperlinks between the pages.

Few properties of massive graphs are

i. Many real world graphs are large. For eg., Facebook graph, contains on an order of a billion vertices and nearly one trillion edges.

ii. They are highly dynamic. New structures appear in social network graphs at every moment. Some of the features of such graphs are new friends can be added, friends can be de-friended.

iii. There is significant interest in the global structure of these graphs, and in particular how that structure changes over time.

iv. We need infinite amount of space and time to solve the properties of these graphs, but having such huge memory is not practical. Compared to the size of the graphs, our computing power is very limited.

There are few every day transactions such as using a phone, using a credit card or browsing a web that leads to storage of data automatically. Hardware technology advancements have made it easier to collect these continuous data. These large volumes of data are then queried to estimate their properties. Massive data is too big to be stored on computer memory of a single machine. Rather, storage devices like tapes are used to store them. So, it is appropriate to assume the input, which is stored in storage devices, to be a stream of data to develop algorithms efficient in time. Since the data is massive, it might not be possible to process the stream by making multiple passes. Sometimes, user is allowed to

process the input elements only once. These conditions, roughly, define the so-called "streaming graph problem", in which the user must analyze a graph presented in the form of either incremental updates or both incremental and decremental updates, using limited space and time. So, one useful model for dealing with massive graphs is the semi-streaming model. The details about data-stream model and the variants of it will be dealt in the next chapters.

The drawback when traditional graph algorithms are applied to graphs which are massive, is the need for them to have a random access to the edges of the graph. The tradeoff between restricted space and the constraints in accessing of input is that at one end, we have dynamic algorithms that may store the whole input graph and in the other, we have polylog(polynomial in logarithm) space restricted streaming algorithms. But, it is very hard to solve the graph problems using only polylog space. It was then suggested by S.Muthukrishnan in [26] that the mid way, ie., algorithms using $\mathcal{O}(n \text{ polylog} n)$ bits of space, is a challenging research area.

1.2 Related Work

Over the past few years, reasonable effort has been done on streaming algorithms for analyzing massive graphs. The data stream model was first proposed by S.Muthukrishnan in [26], [2], [27]. We could refer to [25, 22, 4, 32] for survey on graph stream algorithms. Graph streams can be queried to estimate it's properties as seen in [23, 20, 5]. A lot of work has been done on parameterized complexity and fixed parameter tractability earlier in [16, 17]. The most recent work on undirected feedback vertex set is studied in [29] which gives a quadratic kernel. This result improves the previous kernel bounds given in [6, 7]. The variants of feedback vertex set are discussed in [14, 12].

1.3 Thesis Outline

We start in chapter 2 by giving the background details about data stream model. In Chapter 3, we give the outline about streaming algorithms. In Chapter 4, we discuss the new solutions in semi-streaming model for directed acyclic graphs and matching in turnstile streams. Chapter 5 discusses about parameterized streaming. In Chapter 6, we present the algorithms for finding kernels of edge dominating set and undirected feedback vertex set in parameterized streaming framework.

Chapter 2

Data Stream Algorithms

2.1 Data Stream

Data stream is defined as a continuously arriving sequence of elements drawn from the universe which are accessed in the order. In a stream, data is revealed sequentially, one at a time. Usually streams are massive. Examples of data stream are stream of IP packets, stream of stock prices.

2.2 Data Stream Model

In the classical data stream model([26, 2, 27]), the algorithm processes the stream of data using space which is small when compared to the size of the input. In particular, the algorithm cannot store the whole input and therefore has to construct the summary of the input using less space in order to answer the query. This model can be formalized as a sequence $\sigma = < a_1, a_2, \ldots, a_m >$, where the elements of the stream are drawn from the universe set $[n] := 1, 2, \ldots, n$. m is the stream length and n is the size of the universe.
The goal of the data stream model is to compute a function on the input stream with limited working space, usually which is sublinear in m and n. When we are dealing with massive streams, we may not require exact answer all the time. Even an approximate answer would do the task. Data stream algorithms may even perform pre-processing or post-processing without access to the data stream.

The performance measures of this model are

i. Space utilized by the algorithm.

ii. Number of passes it needs.

iii. Processing time per element([31]).

Input data stream is represented by various models.

i. Cash Register Model : Only data item inserts are possible.

ii. Turnstile Model : Both data item inserts and deletes are possible.

iii. Strict Turnstile Model : Deletion of data item is possible only after it is inserted.

2.3 Techniques used

There are two techniques for handling data streams. The techniques are sampling and sketching. In sampling , we sample few elements from the stream at random and compute a function on these sampled elements. The technique for sampling is given in following algorithm (taken from [26]) :

Algorithm 2.3.1 Sampling Algorithm

Input: A stream a_1 , a_2 , . . . a_n where n is the size of the universe.

1: **procedure** SAMPLING
2: Sample $S = \phi$.
3: **for** each element a_i in stream **do**
4: $S = a_i$ with probability $1/i$.
5: **end for**
6: **end procedure**

Sketching is another useful technique for processing streams. It is a useful summary of the input stream. The basic idea here is to, apply linear projection on the air, that takes the higher dimensional data to a smaller dimensional data in which the properties of graph are preserved and the latter is used to compute any query on the stream. Thus, sketches uses less space and also preserves the relevant properties of original graph. Sketches generally should be

Non-Adaptive : One update on sketch should not be dependent on other.

Loss of independence : The random neighbours returned will not be independent when the sketches are updated repeatedly and queried.

If the input stream is a graph , then sketch looks like this :
Graph sketches project $\mathcal{O}(n^2)$-dimensional graph information into a smaller $\mathcal{O}(polylog\ n)$-dimensional space, the projection preserves structural properties of the graph with high probability:

$$\begin{bmatrix} M \end{bmatrix} \times \begin{bmatrix} v \end{bmatrix} \rightarrow Mv$$

where

a. $M \in \mathbb{R}^{polylogn \times n^2}$ is a large projection matrix, from a particular random family. We do not need to store the whole matrix, it can be computed from a small random seed.

b. $v \in \mathbb{R}^{n^2}$ is a long vector encoding graph information, for example, a row of the incidence matrix.

c. $Mv \in \mathbb{R}^{polylogn}$ is the smaller-dimensional graph sketch which preserves structural properties of v.

Sketches also obey the property of linearity. So,

$$M(u + v) = Mu + Mv \tag{2.1}$$

2.4 Example

Example problems in this model include finding frequency moments, frequent items in stream, heavy hitters in input, etc. We could refer to [8] for these algorithms.

Chapter 3

Streaming Algorithms

3.1 Models in Steaming Algorithms

Streaming algorithms are used for dealing with massive data which arrives as a stream. In this model, elements arrive in a sequence and each element can be read only once. The number of data elements may not be known in advance. Space that can be used is restricted to $(\log n)^{\mathcal{O}(1)}$ where n is the number of data elements.
Variants of streaming model are :

a. **Semi Streaming Model** : This is variant of streaming model, where the space usage is restricted to $\mathcal{O}(n \log n^{\mathcal{O}(1)})$, where n is the number of data elements.

b. **Sliding Window Model** : ([13]) In some applications, we may not require all the data and we might be interested only on new data. One such model is sliding window model in which most recent w elements are taken into account, where w is the length of the window. The elements in the window are said to be active and the rest are said to be expired. An active element may become expired after some point of time but expired elements always remain expired. Only active elements are eligible for estimating statistics or performing the queries. The windows can be sequence-based or time-based windows. Exponential and Smooth histograms are effective methods for estimating queries in this model. An example we can consider is a call-graph where, telephone numbers are represented by nodes and calls placed during some time interval are represented by edges.

c. **W-Stream Model** : This allows the writing of intermediate passes. Intermediate stream at one pass can be used as input stream for the next pass.

d. **Stream-sort Model** : Here, this allows the data stream to be sorted according to the key.

3.2 Graph Streams

State-of-the-art results of graph streams are discussed in [25]. In the previous chapter, we looked at data streams. When the input is a graph, the data stream becomes a graph stream. Massive data could be structured in the form of graphs which leads to the motivation of studying graph streams. Here, the input is a stream that describes a graph and the objective is to estimate various properties of graph. In the graph stream, the input consists of set of edges $(v_i, v_j) \in [n] \times [n]$ where n is the number of vertices and m be the number of edges in graph $G(V, E)$ where $V = \{v_1, \ldots, v_n\}$. Edges arrive in arbitrary order in the stream. The upper bound on m is $\mathcal{O}(n^2)$.
Many large graphs can be represented by a sequence of edges. This sequence of edges may be a stream

only of edge insertions, which gets added to the graph stored, or may be a combination of both edge insertions and deletions. The graph stream can be either fully dynamic or partially dynamic([15]).

Fully Dynamic : Supports insertion and deletion of vertices and edges. Eg : $(u, v, +)$ for insertion, $(u, v, -)$ for deletion. We usually construct sketches of the input to deal with this model.

Partially Dynamic : Supports either insertion or deletion of vertices and edges but not both.

Incremental : Supports only insertions of vertices and edges.

Decremental : Supports only deletions of vertices and edges.

3.2.1 Variants in Graph Stream Model

The following are few variants in graph stream model :

Multi-Pass Models : Here, the stream algorithm is allowed to make more than one pass over the input data. W-Stream and Stream-Sort models can also be used in multi-pass models.

Dynamic, Weighted or Directed graphs : Dynamic graph is a graph which is subject to a sequence of updates. An update can be either edge insertions or edge deletions. For a weighted graph, the stream element will be of the form (u, v, wt) ,where wt is the weight associated with edge (u, v) ie., $v_k \in [n] \times [n] \times R^+$. In a directed graph, an edge (u, v) implies an edge u to v (not vice-versa) in the graph.

Adjacency and Incidence Orderings : In the incidence model, all edges on one vertex arrive consecutively. In the adjacency model any such assumption is not made.

3.3 Example Algorithms

Example problems in graph streams include finding connectivity, bipartiteness, counting triangles, matching, minimum spanning tree, etc in the input stream. Earlier work on these algorithms was discussed in [21]. For the above mentioned standard algorithms in semi-streaming model, we could refer to [8]. Let us look at connectivity algorithm from [8].

Connectivity : An undirected graph is said to be connected , if there exists a path between every pair of vertices in it.
The algorithm is given below.

Algorithm 3.3.1 Connectivity Algorithm

Input: An undirected graph whose stream consists of edges in the form $(u,v) \in [n] \times [n]$ where n is known. but stream length is not known.This is insert only stream.

```
1:  procedure CONNECTIVITY
2:      Initialize F ← φ , i ← 0.
3:      for each element (u, v) in stream do
4:          if ¬i ∧ ((u,v) does not form a cycle in F) then
5:              F = F ∪ (u, v)
6:              if |F| is equal to n − 1 then
7:                  i = 1.
8:              end if
9:          end if
10:     end for
```

11: Output i
12: **end procedure**

Its space usage is $\mathcal{O}(n \log n)$, since size of F is atmost $n-1$ and each edge in F requires $\mathcal{O}(\log n)$ bits.

Chapter 4

New Solutions in Semi-Streaming Model

4.1 Simple Directed Graphs

4.1.1 Preliminaries

Let $G(V, E)$ be a graph where V is the set of vertices and E is the set of edges. Let n be the size of V and m be the size of E. We assume n is known in advance but m is not known. We formally write $G = (V, E)$ where V is a set and $E \subseteq V \times V$. A graph G is said to be undirected , if $\forall$ vertices $u, v \in V$, $(u, v) \in E \Leftrightarrow (v, u) \in E$. Otherwise, G is said to be directed. Input for this problem is a simple dynamic acyclic graph stream which is defined as a $< S = s_1, s_2, ...s_t >$ where $s_k \in [n] \times [n] \times \{-1, 1\}$ defines a multi-graph $G =(V, E)$ where $V = [n]$ and the multiplicity of an edge $e = (u, v)$ is represented by x which is equal to $x(e) = |k : s_k = (u, v, +)| - |k : s_k = (u, v, -)|$. But in our problems, we assume the graph to be simple. The example of dynamic graph stream/turnstile stream is shown in Figures 4.1.1 and 4.1.2. The first part of Figure 4.1.1 is the original graph. The second part is after arrival of the edge (1,2,+). The first part of Figure 4.1.2 is after the arrival of the edge (2, 3, +) and the next part is after the arrival of the edge (1,2,−).

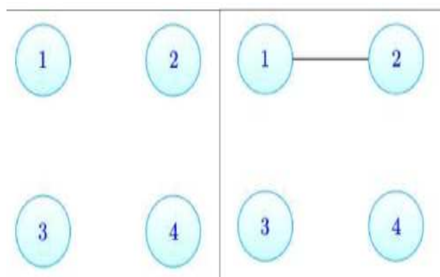

Figure 4.1.1: Turnstile Stream

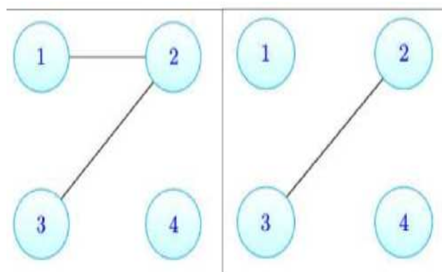

Figure 4.1.2: Turnstile Stream Contd

The following definition and lemma are from [3].

Definition An (ϵ, δ) l_p sampler for $x \neq 0$ returns $\perp$ with probability atmost δ and otherwise returns some $i \in [n]$ with probability in the range :

$$\left[\frac{(1-\epsilon)|x_i|^p}{l_p^p(x)}, \frac{(1+\epsilon)|x_i|^p}{l_p^p(x)}\right]$$

where $l_p(x) = \left(\sum_{i\in[n]} |x_i|^p\right)^{\frac{1}{p}}$

Lemma 4.1.1. *There exists a linear sketch-based algorithms that perform l_p sampling using $\mathcal{O}(\log^2 n \log \delta^{-1})$ space for $p = 0$. We may set $\epsilon = 0$ in this case.*

4.1.2 Algorithms for Simple Directed Graphs

Not much research has been done on directed graphs till now. Yet, many real time graphs are directed. Few examples are

i. Airline route maps : Here airports are vertices and there is an edge from vertex u to vertex v if there is flight from airport u to airport v.

ii. Flowcharts : Here boxes are represented by vertices and arrows are represented by arrows.

iii. Social graphs : Here people are vertices and follow links are edges.

iv. Web graphs : Here web pages are vertices and hyperlinks are edges.

v. One way streets in a map : Here crossings are vertices and streets are edges.

vi. Telephone graphs : Here phones are vertices and calls made are edges.

Let us first look at the connectivity algorithms on directed acyclic graph problems in dynamic graph streams under semi-streaming model. Then, as we can see in [3], all other properties of graph like k-connectivity, bipartiteness, approximate arborescence can be easily computed using connectivity algorithm, given the roots.

4.1.2.1 Connectivity

To check the connectivity of the input directed acyclic graph, we construct minimum spanning tree of it starting with root. All the vertices reachable from the root forms a connected component. To find the minimum spanning tree of the input directed acyclic dynamic graph, we cannot store all the edges of the input stream as it would require a space of $\mathcal{O}(n^2)$. But the space restriction in semi-streaming

model is $\mathcal{O}(n\ polylog\ n)$. So, we use a concept called sketching, where we construct a sketch of size d (where $d << n^2$) for an $\mathcal{O}(n^2)$ - dimensional graph in such a way that the sketch preserves the relevant properties of the original graph with high probability.
We now present a single pass, semi-streaming algorithm that checks the connectivity in the input graph for the given root. The maximum number of edges in directed graph are $n*(n-1)$, where n is the number of vertices. An edge (u, v) in G represents a path from u to v where u is called a head and v is called a tail. A directed spanning tree of G rooted at r is a spanning tree in which no two edges share their tails. Each vertex is the tail of atmost one edge of the directed spanning tree.

Let us see how to construct sketches now.
Given a directed graph G, we construct a sketch matrix where each row in the matrix correspond to a vertex.Formally, we define a $n \times 2\binom{n}{2}$ matrix S_G with entries $(i,(j,k)) \in [n] \times 2\binom{[n]}{2}$ defined by

$$\begin{aligned} S_i(j,k) &= 1\ if\ i = j\ and\ (v_j, v_k)\ \in\ E \\ S_i(j,k) &= -1\ if\ i = k\ and\ (v_j, v_k)\ \in\ E \\ &= 0\ otherwise \end{aligned} \tag{4.1}$$

Each row S_i in S_G corresponds to vertex v_i. So, as we can see, for directed graphs sketches are of form $S_i = (0,1)^{2*({}^nC_2)}$. Given a directed graph, the basic algorithm, given the root(Let the root be u), is as follows :

1. In the first stage, the first phase is as follows
   a. We find an incident edges for all the vertices starting with root.
   b. For each incident edge, if the edge can be added to spanning tree/forest, we merge the vertices of this incident edge into a super-vertex. This in turn has one step.
      i. Let this incident edge be (u, v).
      ii. We remove edges from any other vertex to v as each vertex can become a tail atmost once in a spanning forest.
2. We repeat these steps until the super-vertices cannot be collapsed any further.
3. The minimum spanning tree of the graph is the graph formed by the edges selected. If we get one connected component, we can say that the input graph is connected otherwise disconnected and we can output the number of connected components in the graph for the given root.

The example is shown in Figure 4.1.3. Assume all edges are forward directed edges. This example is taken from [3].

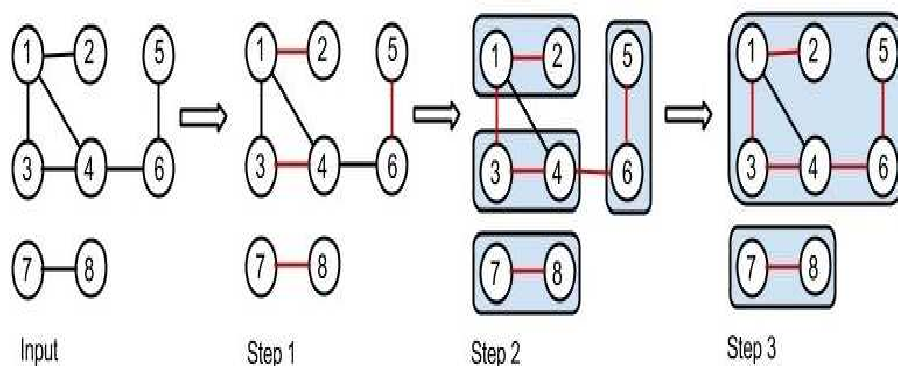

Figure 4.1.3: Spanning Forest Example

Now, we develop a sketch-based algorithm to find the spanning forest and the number of connected components in the input stream. To find an incident edge for any vertex, we could simply l_0 sample the sketch of the vertex. l_0 sampling returns a neighbour of the vertex with uniform probability. Let (u, v) incident on u be the edge selected by l_0 sampler for vertex u. We then merge these two vertices u and v into a super-vertex u, v by adding the sketches of both these vertices. Let the merged sketch be $S_{u,v}$. Since $\{u, v\}$ is a super-vertex, the edges between $\{u\}$ and $\{v\}$ doesn't exist. So we update these entries as 0 in the merged sketch. The next modification which needs to be done to the sketches is that the entry which has a tail as u has to be nullified in each vertex sketch as each vertex can become a tail atmost once. Let X be a sketch which is used to update the sketches of all vertices.

To ensure that every vertex in spanning forest can become a tail atmost once, we construct X as

$$\begin{aligned} X(j,k) &= 0 \ if \ k = v \\ &= 1 \ otherwise \end{aligned} \tag{4.2}$$

We then multiply this sketch X with the every vertex sketch. We can clear X after each stage and reuse it for all stages.

4.1.2.1.1 Sketch-Based Algorithm We maintain $\mathcal{O}(\log n)$ sketch matrices for each vertex which means that , if S_1 is used in first stage of algorithm, then S_2 is used in next stage and so on. Given the roots, Bipartiteness, Approximate arborescence, K-edge connectivity can be found using the same algorithms as given in [3]. The algorithm is shown in Algorithm 4.1.1 :

Algorithm 4.1.1 Connectivity Algorithm in Directed Acyclic Graphs

---

Input: Directed acyclic graph whose stream which consists of edges in the form $(u, v, +/-) \in [n] \times [n] \times \mathbb{R}$ where n is known but stream length is not known and $u < v$ or vice versa.

Input: A root s.

1: **procedure** Connectivity in Directed Graphs

2: Sketch $v_1, v_2, . . , v_n$ using the sketch matrices $S_1, S_2, . ., S_t$ where $t = \mathcal{O}(\log n)$.

3: Initialize the set of supervertices as $V' = V$

4: **for** $r \in [t]$ **do**

5: In the first phase, we find incident edges for all the vertices starting with the root. For the root $s \in V'$, l_0 sample an edge using the sketch $\sum_{v_i \in s} S_r(a_i)$. Let the sampled edge for root be (s, v_i).

6: **for** each edge in incident edges **do**

7: **if** edge can be added to spanning tree **then**

8: Merge the vertices into a super-vertex in V'.

a) Update the entries in all sketches where v_i is a tail as 0 using X.

b) Clear X.

9: **end if**

10: **end for**

11: Repeat the above steps for other vertices which corresponds to the next phases and then proceed to second stage to find incident edges from one super-vertex to other.

12: **end for**

13: The spanning forest of the input graph is formed by the set of sampled edges and G has $|V'|$ number of connected components.

14: **end procedure**

There are $\log n$ sketches for each vertex, so, the space complexity of this algorithm is $\mathcal{O}(n \text{ polylog } n)$ and it returns the spanning forest of the input graph.
Let us consider a small example(with root as 1) as shown in Figure 4.1.4:

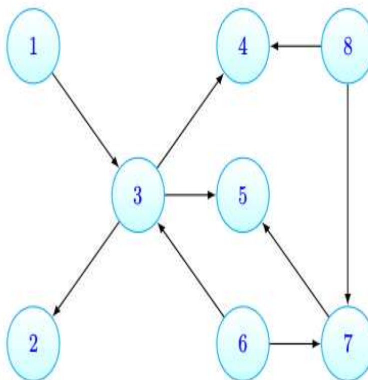

Figure 4.1.4: Spanning Forest : Example 1

We will construct the sketches for the example of Figure 4.1.4 (as shown in Table 4.1.1) :

Table 4.1.1: Sketch Table of connectivity : Step 1

Node	(1,3)	(3,2)	(3,4)	(3,5)	(6,3)	(6,7)	(7,5)	(8,4)	(8,7)
s_1	1	0	0	0	0	0	0	0	0
s_2	0	-1	0	0	0	0	0	0	0
s_3	-1	1	1	1	-1	0	0	0	0
s_4	0	0	-1	0	0	0	0	-1	0
s_5	0	0	0	-1	0	0	-1	0	0
s_6	0	0	0	0	1	1	0	0	0
s_7	0	0	0	0	0	-1	1	0	-1
s_8	0	0	0	0	0	0	0	1	1

So, in the first stage, let the edges sampled be (1, 3), (3, 4), (6, 3), (7, 5), (8, 7).

The first incident edge is the root's edge. (1, 3) edge can be added to spanning forest. All the other edges having 3 as tail are removed from the sketches by multiplying all the sketches with constructed sketch X. Now the step wise sketches look like as shown in Tables 4.1.2, 4.1.3. Now $V' = \{\{1, 3\}, 2, 4, 5, 6, 7, 8\}$.

Table 4.1.2: Sketch Table of connectivity : Step 2

Node	(1,3)	(3,2)	(3,4)	(3,5)	(6,3)	(6,7)	(7,5)	(8,4)	(8,7)
$s_{1,3}$	0	1	1	1	-1	0	0	0	0
s_2	0	-1	0	0	0	0	0	0	0
s_4	0	0	-1	0	0	0	0	-1	0
s_5	0	0	0	-1	0	0	-1	0	0
s_6	0	0	0	0	1	1	0	0	0
s_7	0	0	0	0	0	-1	1	0	-1
s_8	0	0	0	0	0	0	0	1	1

Table 4.1.3: Sketch Table of connectivity : Step 2 - Construction of X

Node	(1,3)	(3,2)	(3,4)	(3,5)	(6,3)	(6,7)	(7,5)	(8,4)	(8,7)
X	0	1	1	1	0	1	1	1	1

After multiplying X with every sketch, Table 4.1.4 shows how the sketches look like after merging the nodes 1 and 3 :

Table 4.1.4: Sketch Table of connectivity : Final Step 2

Node	(1,3)	(3,2)	(3,4)	(3,5)	(6,3)	(6,7)	(7,5)	(8,4)	(8,7)
$s_{1,3}$	0	1	1	1	0	0	0	0	0
s_2	0	-1	0	0	0	0	0	0	0
s_4	0	0	-1	0	0	0	0	-1	0
s_5	0	0	0	-1	0	0	-1	0	0
s_6	0	0	0	0	0	1	0	0	0
s_7	0	0	0	0	0	-1	1	0	-1
s_8	0	0	0	0	0	0	0	1	1

The next edge is (3, 4). This edge can be added to spanning tree as 4 has not become a tail even once till now. We now add the sketches $s_{1,3}$ and s_4 which is shown in Table 4.1.5. Now 4 cannot become a tail again in the spanning tree. So we construct X as shown in Table 4.1.6. Now $V' = \{\{1,3,4\},2,5,6,7,8\}$.

Table 4.1.5: Sketch Table of connectivity : Step 3

Node	(1,3)	(3,2)	(3,4)	(3,5)	(6,3)	(6,7)	(7,5)	(8,4)	(8,7)
$s_{1,3,4}$	0	1	0	1	0	0	0	-1	0
s_2	0	-1	0	0	0	0	0	0	0
s_5	0	0	0	-1	0	0	-1	0	0
s_6	0	0	0	0	0	1	0	0	0
s_7	0	0	0	0	0	-1	1	0	-1
s_8	0	0	0	0	0	0	0	1	1

Table 4.1.6: Sketch Table of connectivity : Step 3 - Construction of X

Node	(1,3)	(3,2)	(3,4)	(3,5)	(6,3)	(6,7)	(7,5)	(8,4)	(8,7)
X	1	1	0	1	1	1	1	0	1

After multiplying X with every sketch, Table 4.1.7 shows how the sketches look like after merging the nodes $\{1,3\}$ and 4 :

Table 4.1.7: Sketch Table of connectivity : Final Step 3

Node	(1,3)	(3,2)	(3,4)	(3,5)	(6,3)	(6,7)	(7,5)	(8,4)	(8,7)
$s_{1,3,4}$	0	1	0	1	0	0	0	0	0
s_2	0	-1	0	0	0	0	0	0	0
s_5	0	0	0	-1	0	0	-1	0	0
s_6	0	0	0	0	0	1	0	0	0
s_7	0	0	0	0	0	-1	1	0	-1
s_8	0	0	0	0	0	0	0	0	1

The next edge is (6, 3). This cannot be added to spanning tree as 3 has already become a tail once. So we don't perform any operation on sketches.

The next edge is (7, 5). This edge can be added to spanning tree as 5 has not become a tail even once till now. We now add the sketches s_7 and s_5 which is shown in Table 4.1.8. Now 5 cannot become a tail again in the spanning tree. So we construct X as shown in Table 4.1.9. Now $V' = \{\{1,3,4\},2,6,\{7,5\},8\}$.

Table 4.1.8: Sketch Table of connectivity : Step 4

Node	(1,3)	(3,2)	(3,4)	(3,5)	(6,3)	(6,7)	(7,5)	(8,4)	(8,7)
$s_{1,3,4}$	0	1	0	1	0	0	0	0	0
s_2	0	-1	0	0	0	0	0	0	0
s_6	0	0	0	0	0	1	0	0	0
$s_{7,5}$	0	0	0	-1	0	-1	0	0	-1
s_8	0	0	0	0	0	0	0	0	1

Table 4.1.9: Sketch Table of connectivity : Step 4 - Construction of X

Node	(1,3)	(3,2)	(3,4)	(3,5)	(6,3)	(6,7)	(7,5)	(8,4)	(8,7)
X	1	1	1	0	1	1	0	1	1

After multiplying X with every sketch, Table 4.1.10 shows how the sketches look like after merging the nodes 7 and 5 :

Table 4.1.10: Sketch Table of connectivity : Final Step 4

Node	(1,3)	(3,2)	(3,4)	(3,5)	(6,3)	(6,7)	(7,5)	(8,4)	(8,7)
$s_{1,3,4}$	0	1	0	0	0	0	0	0	0
s_2	0	-1	0	0	0	0	0	0	0
s_6	0	0	0	0	0	1	0	0	0
$s_{7,5}$	0	0	0	0	0	-1	0	0	-1
s_8	0	0	0	0	0	0	0	0	1

The next edge is (8, 7). This edge can be added to spanning tree as 7 has not become a tail even once till now. We now add the sketches $s_{7,5}$ and s_8 which is shown in Table 4.1.11. Now 7 cannot become a tail again in the spanning tree. So we construct X as shown in Table 4.1.12. Now $V' = \{\{1,3,4\}, 2, 6, \{7,5,8\}\}$.

Table 4.1.11: Sketch Table of connectivity : Step 5

Node	(1,3)	(3,2)	(3,4)	(3,5)	(6,3)	(6,7)	(7,5)	(8,4)	(8,7)
$s_{1,3,4}$	0	1	0	0	0	0	0	0	0
s_2	0	-1	0	0	0	0	0	0	0
s_6	0	0	0	0	0	1	0	0	0
$s_{7,5,8}$	0	0	0	0	0	-1	0	0	0

Table 4.1.12: Sketch Table of connectivity : Step 5 - Construction of X

Node	(1,3)	(3,2)	(3,4)	(3,5)	(6,3)	(6,7)	(7,5)	(8,4)	(8,7)
X	1	1	1	0	1	0	1	1	0

After multiplying X with every sketch, Table 4.1.13 shows how the sketches look like after merging the nodes $\{7,5\}$ and 8 :

Table 4.1.13: Sketch Table of connectivity : Final Step 5

Node	(1,3)	(3,2)	(3,4)	(3,5)	(6,3)	(6,7)	(7,5)	(8,4)	(8,7)
$s_{1,3,4}$	0	1	0	0	0	0	0	0	0
s_2	0	-1	0	0	0	0	0	0	0
s_6	0	0	0	0	0	0	0	0	0
$s_{7,5,8}$	0	0	0	0	0	0	0	0	0

In the next stage, the sampled edges is only (3, 2) as remaining sketches do not have any neighbours. We now add the sketches $s_{1,3,4}$ and s_2 which is shown in Table 4.1.14. Now 2 cannot become a tail again in the spanning tree. So we construct X as shown in Table 4.1.15. Now $V' = \{\{1, 3, 4, 2\}, 6, \{7, 5, 8\}\}$.

Table 4.1.14: Sketch Table of connectivity : Step 6

Node	(1,3)	(3,2)	(3,4)	(3,5)	(6,3)	(6,7)	(7,5)	(8,4)	(8,7)
$s_{1,3,4,2}$	0	0	0	0	0	0	0	0	0
s_6	0	0	0	0	0	0	0	0	0
$s_{7,5,8}$	0	0	0	0	0	0	0	0	0

Table 4.1.15: Sketch Table of connectivity : Step 6 - Construction of X

Node	(1,3)	(3,2)	(3,4)	(3,5)	(6,3)	(6,7)	(7,5)	(8,4)	(8,7)
X	1	0	1	1	1	1	1	1	1

After multiplying X with every sketch, Table 4.1.16 shows how the sketches look like after merging the nodes $\{1, 3, 4\}$ and 2 :

Table 4.1.16: Sketch Table of connectivity : Final Step 6

Node	(1,3)	(3,2)	(3,4)	(3,5)	(6,3)	(6,7)	(7,5)	(8,4)	(8,7)
$s_{1,3,4,2}$	0	0	0	0	0	0	0	0	0
s_6	0	0	0	0	0	0	0	0	0
$s_{7,5,8}$	0	0	0	0	0	0	0	0	0

Now $V' = \{\{1, 3, 4, 2\}, 6, \{7, 5, 8\}\}$.
There are no more neighbours for vertices in V'. So, this algorithm stops after 2 stages and the number of connected components are number of vertices in V'. For root 1, the graph is disconnected and there are three connected components in the example above.

4.2 Matching in Turnstile Streams

4.2.1 Definitions

Let $G(V,\, E)$ be a simple undirected unweighted graph where V is the set of vertices and E is the set of edges. Let n be the number of vertices and m be the number of edges. We assume n is known in advance. We formally write $G = (V,\, E)$ where V is a set and $E \subseteq V \times V$. A matching, M of graph G is a subset of edges of E, such that no two edges in M are adjacent. Maximal matching is a matching, where any edge that is violating the property cannot be added. Maximum matching is a matching with maximum number of edges. The example of maximal and maximum matching is given in Figure 4.2.1.

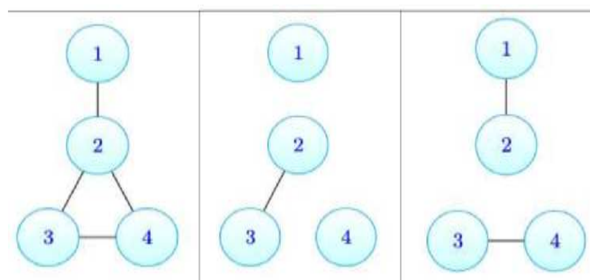

Figure 4.2.1: Matching

As we can see, in the Figure 4.2.1,

First graph is the original graph.

Second graph is maximal matching but not maximum.

Next graph is maximum matching.

4.2.2 Maximal matching in dynamic graph streams

This is one of the open problems as suggested by *Andrew McGregor* in Matching Open Problem. Earlier work on this was discussed in [24] where they gave a multi-pass algorithm for matching, but not in streaming model. Input here is a dynamic graph stream which is defined as $< S = s_1, s_2, ...s_t >$ where $s_k \in [n] \times [n] \times \{-1, 1\}$. Our goal is to find maximal matching in dynamic graph streams under semi-streaming model. It is very well known fact that maximal matching is a 2-approximation of maximum matching whose proof is given below in theorem 4.2.1(referred to 2-appx Proof).

Theorem 4.2.1. *Let M_1 be a maximal matching and M_2 be a maximum matching in an arbitrary graph. Show that maximal matching is a 2-approximation of maximum matching.*

Proof. Since M_2 is maximum, $|M_1| \leq |M_2|$. We now prove that $|M_2| \leq 2.|M_1|$. We go through the edges of M_2 one by one. For any edge $(i, j) \in M_2$, either i is matched in M_1 or j is matched in M_1. Otherwise, this edge could have been added to M_1 . Hence, the matched vertices in M_1 is atleast half of the matched vertices in M_2. Let these sets be denoted by $V(M_1)$, $V(M_2)$. Clearly, $V(M_1) = 2|M_1|$, $V(M_2) = 2|M_2|$, Since now we have that $V(M_2) \leq 2.V(M_1)$, the result is obvious. ☐

We present two solutions for matching in turnstile model, where in one solution, we consider incidence model and in the other, we consider arbitrary input stream.

4.2.2.1 Solution 1

The input here is an incidence stream of edges of a simple undirected unweighted graph $G(V, E)$. In incidence model, all edges incident on one vertex arrive together. So, all insertions and deletions of one vertex appear consecutively. The edges (u, v) in E are, such that $u < v$. An edge can be deleted only after being inserted.
Example of stream in incidence model is
Eg : $\{(1, 2, +)$,$(1, 4, +)$,$(1, 3, +)$,$(1, 2, -)$,$(1, 4, -)$,$(2, 3, +)$,$(2, 4, +)$,$(2, 3, -)$,$(3, 4, +)$,
$(3, 4, -)\}$.
Now in the algorithm, we maintain two buffers M_m and B where M_m is used to maintain maximal matching in G and B is a buffer to hold replacement edges. For the set of incident edges to first vertex, we follow greedy approach for matching and add the edges selected to M_m. Add the remaining insertion edges incident to same vertex to the buffer B. If an edge added to M_m is deleted later in the stream, then we replace this with any possible edge from B. If any edge is a deletion edge and if it is present in B, then delete it from B. After reading the edges incident to one vertex, we can clear the buffer B and use it for the next set of edges.
The algorithm is given below.

Algorithm 4.2.1 Matching in Turnstile Stream Algorithm : Solution 1

Input: An undirected graph whose stream consists of edges in the form $(u, v, +/-) \in [n] \times [n] \times \mathbb{R}$ where n is known but stream length is not known and $u < v$. Stream follows incidence model.

1: **procedure** MATCHING SOLUTION 1

Require: Maximal matching $M_m = \phi$ and buffer $B = \phi$.

```
2:      for all the veritces u ∈ V do
3:          for each edge (u, v) in the stream do
4:              if the edge is an insertion edge (u, v, +) then
5:                  Follow the greedy approach for matching.
6:                  if it is possible to add edge to M_m then
7:                      Add it to M_m.
8:                  else
9:                      Add the edge to B.
10:                 end if
11:             end if
12:             if the edge is a deletion edge (u, v, −) then
13:                 if this edge is in M_m then
14:                     Delete this edge from M_m.
15:                     for the edges e in B do
16:                         if e can be added to M_m then
17:                             Transfer e from B to M_m and break the loop.
18:                         end if
19:                     end for
20:                 end if
21:                 if this edge is in B then
22:                     Delete it from B.
23:                 end if
24:             end if
25:         end for
26:         Clear the buffer B.
27:     end for
28: end procedure
```

4.2.2.1.1 Example : Let us look at a small example now.

The input stream is as follows :

$(1, 2, +)$, $(1, 3, +)$, $(1, 4, +)$, $(1, 2, -)$, $(1, 5, +)$, $(1, 6, +)$, $(1, 5, -)$,
$(2, 3, +)$, $(2, 4, +)$, $(2, 5, +)$, $(2, 4, -)$, $(2, 6, +)$,
$(3, 4, +)$, $(3, 5, +)$, $(3, 6, +)$, $(3, 5, -)$,
$(4, 5, +)$, $(4, 6, +)$, $(4, 5, -)$,
$(5, 6, +)$, $(5, 6, -)$.

1. When $(1, 2, +)$ arrives ,
    i. We follow the greedy approach to maintain maximal matching.
    ii. M will have $(1, 2)$ and buffer is empty set $\{\}$.
2. When $(1, 3, +)$ arrives ,
    i. M still has $\{(1,2)\}$ and buffer contains $\{(1,3)\}$.
3. When $(1, 4, +)$ arrives ,
    i. M still has $\{(1,2)\}$ and buffer contains $\{(1,3),(1,4)\}$.

4. When (1, 2, -) arrives ,
   i. Since edge (1, 2) is deleted , it will be replaced by any edge from the buffer. So M will have $\{(1, 3)\}$ and buffer contains $\{(1, 4)\}$.
5. When (1, 5, +) arrives ,
   i. M has $\{(1, 3)\}$ and buffer contains $\{(1, 4), (1, 5)\}$.
6. When (1, 6, +) arrives ,
   i. M has $\{(1, 3)\}$ and buffer contains $\{(1, 4), (1, 5), (1, 6)\}$.
7. When (1, 5, -) arrives ,
   i. M will have $\{(1, 3)\}$ and buffer contains $\{(1, 4), (1, 6)\}$.

   After this , buffer is reset to empty set.
8. When (2, 3, +) arrives ,
   i. M still has $\{(1, 3)\}$ and buffer contains $\{(2, 3)\}$.
9. When (2, 4, +) arrives ,
   i. M will have $\{(1, 3), (2, 4)\}$ and buffer is $\{\}$.
10. When (2, 5, +) arrives ,
    i. M still has $\{(1, 3), (2, 4)\}$ and buffer is $\{(2, 5)\}$.
11. When (2, 4, -) arrives ,
    i. M will have $\{(1, 3), (2, 5)\}$ and buffer is $\{\}$.
12. When (2, 6, +) arrives ,
    i. M will have $\{(1, 3), (2, 5)\}$ and buffer is $\{(2, 6)\}$.

    After this , buffer is reset to empty set.
13. When (3, 4, +) arrives ,
    i. As 3 is already present in matching. We cannot add the edges incident on 3 to M. So, we neither add edges incident on 3 to M nor B.
14. When (3, 5, +) arrives ,
    i. Ignore this edge.
15. When (3, 6, +) arrives ,
    i. Ignore this edge.
16. When (3, 5, -) arrives ,
    i. Ignore this edge.
17. When (4, 5, +) arrives ,

i. M still has $\{(1,3),(2,5)\}$ and buffer will have $\{(4,5)\}$.

18. When (4, 6, +) arrives ,

    i. M will have $\{(1,3),(2,5),(4,6)\}$ and buffer will have $\{(4,5)\}$.

19. When (4, 5, -) arrives ,

    i. M will have $\{(1,3),(2,5),(4,6)\}$ and buffer is $\{\}$.

20. When (5, 6, +) arrives ,

    i. As 5 is already present in matching. We can ignore all edges incident on 5.

21. When (5, 6, -) arrives ,

    i. Ignore this edge.

Following this approach, maximal matching M will have $\{(1,3),(2,5),(4,6)\}$.

At any time buffer will store atmost n edges if we consider a simple graph. (where n is the number of vertices.)
So, the space complexity is $\mathcal{O}(n)$.

4.2.2.2 Solution 2

The input here is an arbitrary stream of edges of a simple undirected unweighted graph $G(V,\ E)$. The edges $(u,\ v)$ in E are, such that $u < v$. An edge can be deleted only after being inserted.
We maintain maximal matching in M_m. We construct sketches for all the vertices S_i , $\forall i \in V$ in the input stream. We also maintain a buffer U_m to hold unmatched vertices and a buffer V_m for matched vertices. The algorithm explanation is as follows :
For every edge (u, v) in input stream,

a. If (u, v) is an insertion edge,

   1. Then, we add this edge to maximal matching M_m if possible and if (u, v) is added to M_m, then maintain the vertices in this edge in V_m.
   2. If this edge is added to M_m, check if u or v or both is present in U_m, if found, delete them from U_m as U_m maintains only unmatched vertices.
   3. Update sketches S_u , S_v ie., $S_u(u, v) = S_v(u, v) = 1$ to make sure that we have seen this edge in the stream which is not yet deleted.

b. If (u, v) is a deletion edge,

   1. If this edge in present in matching M_m , then delete (u, v) from M_m, update sketches S_u, S_v ie., $S_u(u, v) = S_v(u, v) = 0$(which means either this edge has not arrived in the stream or it is deleted) and delete vertices u and v from V_m.
      i. We first check if all vertices other than u, v are present in V_m. If yes, we need not have to find any replacement edge for both u and v, else l_0 sample S_u until we find a replacement edge for u. We know that l_0 sampling returns a neighbour with equal probability.
      ii. l_0 sample S_v until we find a replacement edge for v.
      iii. If replacement edge is not found for either u or v or both, then store that particular vertex(vertices) in buffer U_m.

iv. In case replacement edges are found for either u or v or both , then the replacement edges are of form (u , w), (v , x). First add the replacement edges found to M_m and the matched vertices to V_m. If any of w or x is there in U_m, delete that particular vertex (or vertices) from U_m.

2. else update sketches S_u , S_v ie., $S_u(u,v) = S_v(u,v) = 0$ which implies that this edge is deleted after being inserted.

After the stream has completely arrived, we find possible replacement edges for each unmatched vertex of U_m and add them to matching M_m. The space utilized is $\mathcal{O}(n + n\ polylog\ n)$.
The algorithm is as given below :

Algorithm 4.2.2 Matching in Turnstile Stream Algorithm : Solution 2

Input: A stream which consists of edges of a simple undirected unweighted graph $G(V, E)$ in the form $(u, v, +/-) \in [n] \times [n] \times \mathbb{R}$ where n is known but m is not known and $u < v$. Stream follows arbitrary model.

1: **procedure** MATCHING SOLUTION 2

Require: Maximal matching $M_m = \phi$, matched vertices buffer $V_m = \phi$, unmatched vertices buffer $U_m = \phi$ and sketches for all vertices S_i , $\forall i \in V$.

2: **for** each edge (u, v) in the stream **do**
3: **if** the edge is an insertion edge $(u, v, +)$ **then**
4: **if** adding (u,v) to maximal matching M_m is possible **then**
5: Add (u,v) is added to M_m.
6: Add the vertices u and v to matched vertices buffer V_m.
7: **if** $u \in U_m$ **then**
8: Delete u from U_m.
9: **end if**
10: **if** $v \in U_m$ **then**
11: Delete v from U_m.
12: **end if**
13: **end if**
14: Update sketches S_u , S_v ie., $S_u(u,v) = S_v(u,v) = 1$.
15: **end if**
16: **if** the edge is a deletion edge $(u, v, -)$ **then**
17: **if** $(u, v) \in M_m$ **then**
18: Delete (u,v) from M_m
19: Update sketches S_u,S_v ie., $S_u(u,v) = S_v(u,v) = 0$
20: Delete vertices u and v from V_m.
21: **if** $|V_m| = n - 2$ **then**
22: Don't find replacement edges.
23: **else**
24: **if** degree of u is $< n \log n$ **then**
25: l_0 sample S_u until we find a replacement edge for u.
26: **else**
27: Sample $(n \log n)$ edges from S_u and find a replacement edge for u in this sample.
28: **end if**
29: **if** degree of v is $< n \log n$ **then**
30: l_0 sample S_v until we find a replacement edge for v.
31: **else**
32: Sample $(n \log n)$ edges from S_v and find a replacement edge for v in this sample.

33: **end if**
34: **end if**
35: **if** replacement edge for u is found (Let it be (u, x)) **then**
36: Add (u, x) to M_m.
37: Add the vertices u and x to matched vertices buffer V_m.
38: **if** $u \in U_m$ **then**
39: Delete u from U_m.
40: **end if**
41: **if** $x \in U_m$ **then**
42: Delete x from U_m.
43: **end if**
44: **else**
45: Add u to U_m.
46: **end if**
47: **if** replacement edge for v is found (Let it be (v, w)) **then**
48: Add (v, w) to M_m.
49: Add the vertices v and w to matched vertices buffer V_m.
50: **if** $v \in U_m$ **then**
51: Delete v from U_m.
52: **end if**
53: **if** $w \in U_m$ **then**
54: Delete w from U_m.
55: **end if**
56: **else**
57: Add v to U_m.
58: **end if**
59: **end if**
60: **else**
61: Update sketches S_u , S_v ie., $S_u(u, v) = S_v(u, v) = 0$.
62: **end if**
63: **end for**
64: **for** each vertex in U_m **do**
65: Find the replacement edges and add them to M_m and remove the vertices of these edges from U_m.
66: **end for**
67: **end procedure**

4.2.2.2.1 Example : Let us consider a small example :

The input stream is as follows :

(1, 2, +) , (3, 4, +) , (1, 2, −) , (2, 4, +) , (1, 3, +) , (2, 4, −) ,
(2, 3, +) , (1, 4, +) , (2, 3, −) , (2, 3, +) , (2, 4, +) , (3, 4, −) , (3, 4, +).

1. On seeing the edge (1, 2, +),

   i. $M_m = \{(1, 2)\}$.

ii. $V_m = \{1, 2\}$.

iii. Look at the Table 4.2.1.

iv. $U_m = \phi$.

Table 4.2.1: Matching Sketches Example : 1

Node	12	13	14	23	24	34
s_1	1	0	0	0	0	0
s_2	1	0	0	0	0	0
s_3	0	0	0	0	0	0
s_4	0	0	0	0	0	0

2. On seeing the edge (3, 4, +),

i. $M_m = \{(1, 2), (3, 4)\}$.

ii. $V_m = \{1, 2, 3, 4\}$.

iii. Look at the Table 4.2.2.

iv. $U_m = \phi$.

Table 4.2.2: Matching Sketches Example : 2

Node	12	13	14	23	24	34
s_1	1	0	0	0	0	0
s_2	1	0	0	0	0	0
s_3	0	0	0	0	0	1
s_4	0	0	0	0	0	1

3. On seeing the edge (1, 2, $-$),

i. $M_m = \{(3, 4)\}$.

ii. Look at the Table 4.2.3.

iii. $V_m = \{3, 4\}$.

iv. There are no neighbours for either 1 or 2. So, add them to U_m. Therefore, $U_m = \{1, 2\}$.

Table 4.2.3: Matching Sketches Example : 3

Node	12	13	14	23	24	34
s_1	0	0	0	0	0	0
s_2	0	0	0	0	0	0
s_3	0	0	0	0	0	1
s_4	0	0	0	0	0	1

4. On seeing the edge (2, 4, +),

i. $M_m = \{(3, 4)\}$.

ii. $V_m = \{3, 4\}$.

iii. Look at the Table 4.2.4.

iv. $U_m = \{1, 2\}$.

Table 4.2.4: Matching Sketches Example : 4

Node	12	13	14	23	24	34
s_1	0	0	0	0	0	0
s_2	0	0	0	0	1	0
s_3	0	0	0	0	0	1
s_4	0	0	0	0	1	1

5. On seeing the edge (1, 3, +),

    i. $M_m = \{(3, 4)\}$.

    ii. $V_m = \{3, 4\}$.

    iii. Look at the Table 4.2.5.

    iv. $U_m = \{1, 2\}$.

Table 4.2.5: Matching Sketches Example : 5

Node	12	13	14	23	24	34
s_1	0	1	0	0	0	0
s_2	0	0	0	0	1	0
s_3	0	1	0	0	0	1
s_4	0	0	0	0	1	1

6. On seeing the edge (2, 4, $-$),

    i. $M_m = \{(3, 4)\}$.

    ii. $V_m = \{3, 4\}$.

    iii. Look at the Table 4.2.6.

    iv. $U_m = \{1, 2\}$.

Table 4.2.6: Matching Sketches Example : 6

Node	12	13	14	23	24	34
s_1	0	1	0	0	0	0
s_2	0	0	0	0	0	0
s_3	0	1	0	0	0	1
s_4	0	0	0	0	0	1

7. On seeing the edge (2, 3, +),

    i. $M_m = \{(3, 4)\}$.

    ii. $V_m = \{3, 4\}$.

    iii. Look at the Table 4.2.7.

    iv. $U_m = \{1, 2\}$.

Table 4.2.7: Matching Sketches Example : 7

Node	12	13	14	23	24	34
s_1	0	1	0	0	0	0
s_2	0	0	0	1	0	0
s_3	0	1	0	1	0	1
s_4	0	0	0	0	0	1

8. On seeing the edge (1, 4, +),

   i. $M_m = \{(3, 4)\}$.

   ii. $V_m = \{3, 4\}$.

   iii. Look at the Table 4.2.8.

   iv. $U_m = \{1, 2\}$.

Table 4.2.8: Matching Sketches Example : 8

Node	12	13	14	23	24	34
s_1	0	1	1	0	0	0
s_2	0	0	0	1	0	0
s_3	0	1	0	1	0	1
s_4	0	0	1	0	0	1

9. On seeing the edge (2, 3, −),

   i. $M_m = \{(3, 4)\}$.

   ii. $V_m = \{3, 4\}$.

   iii. Look at the Table 4.2.9.

   iv. $U_m = \{1, 2\}$.

Table 4.2.9: Matching Sketches Example : 9

Node	12	13	14	23	24	34
s_1	0	1	1	0	0	0
s_2	0	0	0	0	0	0
s_3	0	1	0	0	0	1
s_4	0	0	1	0	0	1

10. On seeing the edge (2, 3, +),

   i. $M_m = \{(3, 4)\}$.

   ii. $V_m = \{3, 4\}$.

   iii. Look at the Table 4.2.10.

   iv. $U_m = \{1, 2\}$.

Table 4.2.10: Matching Sketches Example : 10

Node	12	13	14	23	24	34
s_1	0	1	1	0	0	0
s_2	0	0	0	1	0	0
s_3	0	1	0	1	0	1
s_4	0	0	1	0	0	1

11. On seeing the edge (2, 4, +),

   i. $M_m = \{(3, 4)\}$.

   ii. $V_m = \{3, 4\}$.

   iii. Look at the Table 4.2.11.

iv. $U_m = \{1, 2\}$.

Table 4.2.11: Matching Sketches Example : 11

Node	12	13	14	23	24	34
s_1	0	1	1	0	0	0
s_2	0	0	0	1	1	0
s_3	0	1	0	1	0	1
s_4	0	0	1	0	1	1

12. On seeing the edge (3, 4, −),

    i. $M_m = \{(1,3),(2,4)\}$.

    ii. $V_m = \{1,2,3,4\}$.

    iii. Look at the Table 4.2.12.

    iv. $U_m = \phi$.

Table 4.2.12: Matching Sketches Example : 12

Node	12	13	14	23	24	34
s_1	0	1	1	0	0	0
s_2	0	0	0	1	1	0
s_3	0	1	0	1	0	0
s_4	0	0	1	0	1	0

13. On seeing the edge (3, 4, +),

    i. $M_m = \{(1,3),(2,4)\}$.

    ii. $V_m = \phi$.

    iii. Look at the Table 4.2.13.

    iv. $U_m = \phi$.

Table 4.2.13: Matching Sketches Example : 13

Node	12	13	14	23	24	34
s_1	0	1	1	0	0	0
s_2	0	0	0	1	1	0
s_3	0	1	0	1	0	1
s_4	0	0	1	0	1	1

Now, the entire stream is over. Since there are no vertices in U_m, we need not have to find any replacement edges. So the maximal matching is $M_m = \{(1,3),(2,4)\}$.

Theorem 4.2.2. *If the vertex y has degree greater than $n \log n$, then there is a high probability that we find a replacement edge from $n \log n$-recovery sketch of S_y for y.*

Proof. Let $(y,\ x_1)$, $(y,\ x_2)$, . . .$(y,\ x_{n \log n})$ be the edges sampled from sketch S_y. Let N_y be the neighbourhood of vertex y. Let $N_y^{'}$ be the neighbourhood of vertex y at current time instance. Let V_m be the set of matched vertices. Number of matched vertices can be atmost $n-2$ as we are trying to find a replacement edge for an edge of 2 vertices. So, these 2 vertices are not present in V_m. Therefore,

$N_y \cap V_m \leq n-2$ and $N_y - N_y^{'} \leq n-2$.
The probability that there is a matched vertex in neighbourhood of the high degree vertex is

$$\begin{aligned} Pr\Big[x_i \in V_m\Big] &\leq \sum_{z \in N_y' \cap V_m} Pr\Big[x_i = z\Big] \\ &\leq \sum_{z \in N_y' \cap V_m} \frac{1}{{N_y}'} \\ &\leq \sum_{z \in {N_y}' \cap V_m} \frac{1}{N_y - (n-2)} \\ &\leq \frac{n-2}{n \log n - (n-2)} \\ &\leq \frac{1}{\log n} \end{aligned}$$

So, the probability that there is an unmatched vertex in neighbourhood of the high degree vertex is $1 - \frac{1}{\log n}$ □

Chapter 5

Parameterized Complexity Concepts

As mentioned in [18], most of the graph optimization problems are NP-Hard, indicating that, unless P = NP, there can be no polynomial time algorithm which can solve all the instances of an NP-Hard problem exactly. The traditional way to prevent this intractability is to design approximation algorithms where we get an approximate solution or randomized algorithms where we lose the guarantee that the output is always correct. They run in polynomial time. All problems in NP have trivial exponential time algorithms which just search and verify all the witnesses. So, any algorithm which beats this brute-force algorithm is thought of as making a clever search in the big space of all candidate solutions.

5.1 Parameterized Complexity

Parameterized Complexity is essentially a two-dimensional analogue of P vs NP. Here, instead of expressing the running time only in terms of input size, one or more parameters of the input are defined, and we analyze how parameters are effecting the running time. If the input size is huge and if it has a small parameter, the goal of PC is to design efficient algorithms.

In parameterized complexity problem specification has three parts :
1. The input.
2. The parameter(of the input).
3. The statement of the problem.

Definition A parameter is a function from problem instances to the set of natural numbers.([1])

The parameter can be input size, size of optimal solution of the problem, etc.

5.2 Fixed Parameter Tractability

It is not likely that polynomial time algorithms exist for NP Hard and NP Complete problems (FPT_Problems). But many of these problems might be associated with some parameters related to their complexity. If the parameter is small, then there is a hope that we may find polynomial time solution to that instance. This is the basic idea behind Fixed Parameter Tractability. For more information on fixed parameter tractability, we could refer to [17].

Definition We say that a parameterized problem is fixed parameter tractable (FPT) with respect to parameter k if there exists a solution running in $f(k) \times n^{\mathcal{O}(1)}$ time, where f is a function of k which is independent of n.([1])

5.2.1 Techniques used

There are some techniques which can be used to say that a problem is fixed parameter tractable. Few of them are bounded search trees, kernelization, iterative compression and color coding.

5.2.1.1 Bounded Search Trees

This technique does exhaustive search on the problem space. It uses the parameter to branch on the search tree so that running time is still polynomial in input size. For an example, we could refer to [1].

5.2.1.2 Kernelization

The formal definition is

Definition Let $L \subseteq \sum^* \times \sum^*$ be a parameterized language. A reduction to a problem kernel, or kernelization, comprises replacing an instance (I,k) by a reduced instance (I', k'), called a problem kernel, such that

(i) $k' \leq k$.

(ii) $|I'| \leq g(k)$, for some function g depending only on k, and

(iii) $(I, k) \in L$ iff $(I', k') \in L$. ([11])

If a problem is fixed parameter tractable, then we can reduce an instance of size n to an instance of size $f(k)$, which is called as kernel where $f(k)$ is a function in k. A kernelization algorithm applied to some problem instance takes polynomial time in the input size and always returns an equivalent instance (i.e., the instances will have the same answer) of size bounded by a function of some problem-specific parameter.
The formal definition is

Definition A kernelization algorithm (kernel) for a parameterized problem $Q \subseteq \sum^* \times N$ is an algorithm that, for input $(x, k) \in \sum^* \times N$ outputs a pair $(x', k') \in \sum^* \times N$ in $(|x| + k)^{\mathcal{O}}(1)$ time such that $|x'|$, $k' < g(k)$ for some computable function g, called the size of the kernel, and $(x, k) \in Q \iff (x', k') \in Q$. A polynomial kernel is a kernel with polynomial size.

To obtain a kernel, we use some reduction rules that delete the unnecessary parts of the input. In parameterized complexity theory, it is possible to prove that kernel size can be found in polynomial time. Then, FPT algorithm can be applied on this kernel. Infact, a problem is said to be in FPT iff it is kernelizable. For some problems, we can find a kernel ie, a smaller instance of the original instance such that solution to smaller instance is the solution to original instance. The size of the kernel has to be bounded in terms of a function in k , where k is the parameter of the instance.

Now let us consider an example, Vertex Cover with parameter k being the size of optimal solution. Every vertex with degree greater than k has to be in the solution set, because otherwise, all it's neighbours which are more than k will be in the solution set , which is clearly not possible. So, include the vertex with degree greater than k in solution set, remove this vertex and it's incident edges from the graph and repeat the procedure. So, by this point, every vertex has a degree with degree $\leq k$.Thus, our kernel is

found with atmost k^2 edges and atmost $2k^2$ vertices. Now we can apply bounded search tree algorithm on this kernel to get the final solution. The time complexity is $\mathcal{O}(2^k k^2)$. The major part of this information is taken from Kernelization Wiki. The kernel of a graph/dynamic graph can sometimes be found using sampling as shown in [9].

5.3 Parameterized Streaming

In many real world applications, there are instances of graph problems whose solutions are small compared to the size of input. There are some cases where a less number of X meet the need for huge number of Y. For example, very few railway stations can cover the whole city. In these cases, assuming the number of facilities to be a small number k is practical. Parameterized Streaming is a new approach to handle graph streams, where the goal is to seek solutions for the parameterized versions of the graph problems using streaming space restriction. In the parameterized versions of graph problems, each graph problem is associated with a parameter k and the objective is to decide whether there exists solution bounded by k, using a space complexity bounded in terms of k ie., sublinear in the size of the input. The various input streams that could be considered are insert-only stream and dynamic stream. Not much research has been done in this model. Some recent work is discussed in [10, 11, 19].

Chapter 6

New Solutions in Parameterized Streaming

6.1 Preliminaries

Let $G(V, E)$ be a simple undirected graph where V is the set of vertices and E be the set of edges. Let n be the size of V and m be the size of E. We assume that n is known and m is not known. Let $V(E)$ be the set of vertices that are incident with edges in E. For a graph G, let $G(V)$ be the subgraph of G induced by vertices in V. Let $G(E)$ be the subgraph of G induced by edges in E. $G(E) = G(V(E), E)$. If there exists a vertex x in a graph, such that there exist a set of k cycles pairwise intersecting exactly on x, then this is called an x-flower of order k.
A streaming kernelization algorithm (streaming kernel) is an algorithm that receives input (G, k) for a parameterized problem in the following fashion. The algorithm is presented with an input stream where edges of G are presented in a sequence, ie., adhering to the cash register model. Finally, the algorithm should return a kernel for the problem upon request. A t-pass streaming kernel is a streaming kernel that is allowed t passes over the input stream before a kernel is requested. We assume that a streaming kernelization algorithm receives parameter k and the size of the vertex set before the input stream. In the strict streaming kernelization setting the streaming kernel must use at most $p(k) \log |n|$ space where p is a polynomial. In the semi-streaming kernelization setting the streaming kernel must use at most $np(k) \log |n|$ space where p is a polynomial.

6.2 Edge Dominating Set

6.2.1 Definition

Given a graph $G(V, E)$ and an integer k, the EDS problem is to decide whether there exists a set S of atmost k edges such that every edge in $E - S$ is incident with an edge in S. Since we follow a streaming model here, input graph arrives as a sequence of edges either only insertions or both insertions and deletions. The approach we follow here is to find a kernel for edge dominating set, on which a FPT algorithm is run later to get the final solution.

6.2.2 Existing Work

We could refer to [19] for the existing work where they proved that single pass streaming kernel is not possible for edge dominating set. So, they presented a 2-pass kernel which works only in insert-model.

[30] discusses the parameterized solution of edge dominating set.

6.2.2.1 Single Pass Kernel

Theorem 6.2.1. *A single pass streaming kernelization algorithm for Edge Dominating Set(k) requires at least m - 1 bits of local memory for instances with m edges.(from [19])*

Proof. Consider the following instance. Let $G(V,\ E)$ be a simple graph, where $V \subseteq [n]$ and $E = (a, v_i), \forall i \in [n]$. Let A be a single pass streaming kernelization algorithm for Edge Dominating Set with parameter k. Let $k = 1$ for this instance. Let $V = v_1, v_2, ..., v_{|V|}$ be a partial stream which consists of edges $E = (a, v_1), (a, v_2), ..., (a, v_{|V|})$. If A uses less than n bits of memory, then by pigeonhole principle there must exist two partial streams $V', V'' \in [n]$ such that $V' \neq V''$ and both of them result in same kernel. Now let us check whether there exists a pair of streams $< V', e >$ and $< V'', e >$, for every edge e, that result in same kernel. Let us assume w.l.o.g $V' - V'' \neq \phi$ and let $i \in V' - V''$. So, (a, v_i) is an edge dominating set for $E(V') \cup (b, v_i)$, where $b \neq a$. But $E(V'') \cup (b, v_i)$ results in an edge dominating set with two edges, which results a NO instance because $k = 1$. Thus, A does not result in same kernel for both instances, which is a contradiction. Thus any algorithm for edge dominating set uses n bits for an instance on $n + 1$ edges. Thus, for m edges, algorithm requires $m - 1$ bits of memory. ☐

6.2.2.2 2 - Pass Kernel

The algorithm they presented is as follows :

i. In the first pass, find the 2k vertex cover for the input graph.

ii. In the second pass, find the induced graph for vertices stored in first pass.

Space used here is $\mathcal{O}(k^3 \log k)$.

6.2.3 Proposed Work

Let us consider a simple undirected graph whose edges arrive as our input stream. Our algorithm is a 2 pass algorithm which works for both insert and insert-delete streams. Algorithm returns a linear kernel after 2 passes, on which, any FPT algorithm is run to get the final solution. The brief algorithm is as follows :

i. In the first pass, we find the maximal matching of the input graph. Let it be M.

ii. If $|M| \leq k$, output M as the required edge dominating set and quit.

iii. If $|M| > 2k$, then there cannot exist an edge dominating set of size atmost k. So, we return a NO instance.

iv. In the second pass, kernel G' is the induced graph $G(V(M))$ along with one edge for each matched vertex where this edge has one vertex as this matched vertex and the other vertex is any unmatched vertex in it's neighbours.

v. If there is a x-flower of order $\geq k + 1$ in G', then return a NO instance.

In the first pass, we find maximal matching using greedy algorithm for insert streams and using algorithm explained in chapter 4 for dynamic streams. Let it be M. Every maximal matching is an edge dominating set but not every edge dominating set is a matching. So, if the size of M is less than or equal to k, we return M as our edge dominating set. The vertex set of edge dominating set must include atleast one vertex of each edge in maximal matching. So, if the size of maximal matching is greater than

$2k$ which implies the vertex set of the edge dominating set includes atleast $(2k + 1)$ vertices and thus, there do not exist an edge dominating set of size atmost k. So, the size of maximal matching lies between k and $2k$: $k + 1 \leq |M| \leq 2k$ where $|M|$ is the size of matching M. Let matched vertices V_m be the vertices in M and let U_m be the unmatched vertices.
In the second pass, the induced graph on $V(M)$ along with adding one edge for each matched vertex to graph G' where this edge has one vertex as this matched vertex and the other vertex is any unmatched vertex in it's neighbours. If there is a x-flower of order greater than or equal to $k + 1$, then there cannot be an edge dominating set of size atmost k, so, we return a NO instance. Otherwise our kernel is the graph G'. In insert only model, we can greedily add the edges for each matched vertex whose neighbour is unmatched whereas in dynamic model , we could l_0 sample the sketches of matched vertices, that were constructed during the first pass for finding maximal matching, to return unmatched neighbour for each matched vertex. A k-sample recovery algorithm recovers k neighbours from S_x(or all neighbours if number of non-zero entries in $S_x < k$) such that sampled edge (i, j) has $S_x(i, j) \neq 0$ and is sampled uniformly.

Since $k + 1 \leq |M| \leq 2k$ implies $2k + 2 \leq |V(M)| \leq 4k$. All unmatched vertices have neighbours only in matched vertices. So number of unmatched vertices is equal to number of matched vertices which is atmost $4k$. The number of vertices in kernel G' are

$$\begin{aligned} &= V_m + U_m \\ &= 4k + 4k \\ &= 8k \\ &= \mathcal{O}(k) \end{aligned} \tag{6.1}$$

The number of edges in kernel G' are $\mathcal{O}(k^2)$.

Let us prove that this algorithm is safe.

Theorem 6.2.2. *S is an edge dominating set of size atmost k for G iff there is an edge dominating set S' of size atmost k for G'.*

Proof. Let us first prove the forward direction ie., if S is an edge dominating set of size atmost k for G then there exists an edge dominating set S' of size atmost k for G'. We shall construct S' which is initially ϕ.
For every edge (u, v) in S, there exists two cases :

i If $(u, v) \in G'$, then this edge can be added to S'. It covers all the neighbouring edges.

ii If $(u, v) \notin G'$, this means that one of the vertex in (u, v) is a matched vertex and the other is an unmatched vertex. Both of them cannot be unmatched vertices because if they were, this edge would have been added to maximal matching in the first pass. Let the matched vertex be u and the unmatched vertex be v. S will have an edge that covers the matched vertex edge(edge on u in maximal matching) of maximal matching and the edge between this matched vertex and it's unmatched neighbours. So this edge covers (u, v). So, this edge can be covered by either including the matched vertex edge of maximal matching or the edge between this matched vertex and it's unmatched neighbour vertex included in G'.

iii In this way, S' has size atmost k.

Now let us prove the backward direction ie., if there is an edge dominating set S' of size atmost k for G' then S' is an edge dominating set for G too. The only edges that are present in G and not present in G' are some edges between unmatched vertices and matched vertices. But one edge for each matched

vertex with a neighbouring unmatched vertex is added to G'. The remaining edges which are not added to G' are covered by S'. So, S' of size atmost k is also an edge dominating set for G. Thus G has an edge dominating set $S = S'$ of size atmost k. ☐

6.2.3.1 Multi Pass Kernel for Insert Only Stream

Algorithm 6.2.1 Edge Dominating Set Streaming Kernel Algorithm : Insert Only Stream

Input: An undirected graph whose stream consists of edges in the form $(u, v) \in [n] \times [n]$ where n is known but stream length is not known and $u < v$ and, an integer k.

1: **procedure** EDS INSERT-ONLY STREAM

Require: Maintain buffer $M = \phi$, matching vertices $V_m = \phi$ and unmatched vertices $U_m = V - V_m$.

2: ***Pass 1***:
3:     **for** the edge (u, v) in the input stream **do**
4:         **if** $u \in M$ or $v \in M$ **then**
5:             Ignore this edge and don't add this edge to M.
6:         **else**
7:             Add this edge to the maximal matching M.
8:         **end if**
9:     **end for**
10:     **if** $|M| < k$ **then**
11:         Return M as the edge dominating set solution.
12:     **else**
13:         **if** $|M| > 2k$ **then**
14:             Return a NO instance.
15:         **else**
16:             Let $V_m = V(M)$. Go to Pass 2.
17:         **end if**
18:     **end if**
19: ***Pass 2***:
20:     **for** the edge (u, v) in the input stream **do**
21:         **if** both $u \in M$ and $v \in M$ but $(u, v) \notin M$ **then**
22:             Add (u, v) to M.
23:         **else if** $u \notin M$ and $v \in M$ and v is not marked **then**
24:             Add this edge to M.
25:             Mark v as done.
26:         **else if** $u \in M$ and $v \notin M$ and u is not marked **then**
27:             Add this edge to M.
28:             Mark u as done.
29:         **else**
30:             Don't add this edge.
31:         **end if**
32:     **end for**
33:     Kernel $G' = M$.
34:     **if** there is a x-flower of order $\geq k + 1$ in G' **then**
35:         Return a NO instance.
36:     **else**
37:         Output G' as kernel.
38:     **end if**

39: **end procedure**

6.2.3.2 Multi Pass Kernel for Insert Delete Stream

Algorithm 6.2.2 Edge Dominating Set Streaming Kernel Algorithm : Dynamic Streams

Input: An undirected graph whose stream consists of edges in the form $(u, v, +/-) \in [n] \times [n] \times \mathbb{R}$ where n is known but stream length is not known and $u < v$, and an integer k. An edge can be deleted only after being inserted.

1: **procedure** EDS INSERT-DELETE STREAM

Require: Maintain buffer $M = \phi$, matching vertices $V_m = \phi$ and unmatched vertices $U_m = V - V_m$.

2: ***Pass 1***:
3: Refer Maximal matching algorithm in turnstile streams from Algorithm 4.2.1. Let it be M.
4: **if** $|M| < k$ **then**
5: Return M as the edge dominating set solution.
6: **else**
7: **if** $|M| > 2k$ **then**
8: Return a NO instance.
9: **else**
10: Let $V_m = V(M)$. Go to Pass 2.
11: **end if**
12: **end if**
13: ***Pass 2***:
14: **for** the edge (u, v) in the input stream **do**
15: **if** the edge is an insertion edge and both $u \in M$ and $v \in M$ but $(u, v) \notin M$ **then**
16: Add (u, v) to M.
17: **else**
18: **if** the edge is a deletion edge and $(u, v) \in M$ **then**
19: Delete (u, v) from M.
20: **end if**
21: **end if**
22: **end for**
23: **for** each vertex u_m in V_m **do**
24: Sample n edges from the sketch S_{u_m} and add it's incident edge whose other vertex is unmatched to buffer M.
25: **end for**
26: Kernel $G' = M$.
27: **if** there is a x-flower of order $\geq k + 1$ in G' **then**
28: Return a NO instance.
29: **else**
30: Output G' as kernel.
31: **end if**
32: **end procedure**

6.3 Undirected Feedback Vertex Set

6.3.1 Definition

Given an undirected graph $G(V,\, E)$ and an integer k, the FVS problem is to decide whether there exists a set $V' \subseteq V$ such that $G - V'$ has no cycles. In a parameterized semi-streaming model, input graph arrives as a stream of edges and the space restriction is atmost $\mathcal{O}(n\ p(k)\ \log|n|)$ space where $p(k)$ is a polynomial in k. [29] gives an algorithm for finding quadratic kernel(not in the streaming model) which improves the bounds in [6] and [7]. Application of feedback vertex set is deadlocks.

We present two solutions for undirected feedback vertex set with semi-streaming space restriction.

6.3.2 Solution 1

A streaming algorithm may have three parts : pre-processing, processing the data and post processing. The following is the semi-streaming kernel algorithm for UFVS.

Algorithm 6.3.1 Undirected Feedback Vertex Set : Solution 1

1: ***Pass 1***:

Input: A stream which consists of edges of a simple undirected unweighted graph $G(V,\, E)$ in the form $(u,\, v) \in [n] \times [n]$ where n is known but m is not known and $u < v$. Stream follows arbitrary model.

Input: A parameter k.

Require: Maintain minimum spanning tree $T = \phi$ and a count for number of edges $c = 0$.

2: **for** every edge $(u,\, v)$ in the stream **do**

3: $c = c + 1$.

4: **if** Adding this edge to T does not result in a cycle **then**

5: Add this edge to T.

6: **else**

7: Find the path from u to v which resulted in the cycle and write this path in the order to W-stream.(This W-stream is used as input stream for next pass.)

8: **end if**

9: **end for**

10: **if** $c > n + nk$ **then**

11: Return a NO instance.

12: **else**

13: Go to Pass 2.

14: **end if**

15: Space usage for this pass is $\mathcal{O}(n)$.

16: ***Pass 2***:

Input: The W-stream from the previous pass and the parameter k.

Require: Graph $G' = \phi$ where G' has vertex set V' and edge set E'.

Require: Parameter $k' = k$ for G'.

Require: Feedback vertex set $S' = \phi$.

17: **for** every path $(u,\, a,\, b,\, .\, .\, .\, ,\, z,\, v)$ in the W-stream **do**

18: Add edges $(u,\, a),\, (a,\, b),\, .\, .\, .\, .\, ,\, (z,\, v),\, (v,\, u)$ to G'.

19: **end for**

20: Space usage for this pass is $\mathcal{O}(nk)$.

21: ***Post Processing***:

22: By this point degree of every vertex in G' is greater than 2.

23: Now we will apply the reduction rules.
24: ***Reduction Rule 1***:
25: If any vertex w in G' has degree $> 2k + 1$, then

a. $G' = G' - w$.

b. $k' = k' - 1$.

c. $S' = S' \cup w$.

26: ***Reduction Rule 2***:
27: If the degree of any vertex w is ≤ 1, then

a. $G' = G' - w$.

b. S' and k' remains unchanged.

28: Apply the above rules repeatedly until none of the above rules are valid on G'.
29: ***Reduction Rule 3***:
30: **if** there are more than k' connected components in G' **then**
31: Return a NO instance.
32: **else**
33: **if** there exists a set of vertices $X \subseteq V'$ in such a way that $G' - X$ is a spanning forest(F') with maximum number of edges **then**
34: Let x be the maximum degree vertex in F'. Removal of x in F' results in many connected components in F'.
35: Remove the bridge edges between x and every connected component in F'. The only modification made to G' is on x.
36: Add double edges from x to every vertex in X.
37: **end if**
38: **end if**
39: ***Reduction Rule 4***:
40: Apply reduction rule 2 on $G' - x$. So, every vertex in resulting graph has degree ≥ 2.
41: **for** every vertex $y \in X$ **do**
42: **if** there is a path from y to y in $G' - x$ with the internal vertices from $V' - X$ **then**
43: Add a self-loop on y.
44: **end if**
45: **end for**
46: Now our kernel $G'' = G'[X \cup x]$
47: Output (G'', k', S') as the kernel instance.
48: The number of vertices in kernel $= \mathcal{O}(2k')$ and the number of edges $= \mathcal{O}(k'^2)$.

The algorithm explanation is as follows : The definition of UFVS states that we need to find a set of vertices that hits all the cycles in the graph. So, our first pass should be finding all needed cycles from the graph. For finding the cycles, we construct the minimum spanning tree, on the fly, and if any edge(say (u, v)) is forming a cycle , we find the shortest path from u to v in spanning tree and write this path in the order to W-stream. In streaming algorithm, the space to be used is limited. Since we cannot store all cycles formed in any storage, we use the concept of W-stream and write it to W-stream which can be used as input to next pass.

Let S of size atmost k be the feedback vertex set of input graph G. We will now prove that the upper bound on the number of edges in G is $n + nk$. If S is feedback vertex set of $G(V, E)$, then $G(V - S)$ forms a spanning forest which has atmost $|V - S| - 1$ edges. Vertices in S have atmost $(n - 1)|S|$ edges.

So the total number of edges

$$
\begin{aligned}
&= (n-1)|S| + |V-S| - 1 \\
&= (n-1)k + n - k - 1 \\
&= (nk - k) + n - k - 1 \\
&= nk + n - 2k - 1 \\
&< nk + n
\end{aligned}
\tag{6.2}
$$

The space usage is $\mathcal{O}(nk)$. By this time, every vertex in our graph has a minimum degree of 2 which means every vertex is involved in some or the other cycle. In the second pass, we re-construct our graph from cycles of W-stream. Now we will look at the reduction rules and prove that each rule is safe.

Reduction Rule 1 : If any vertex has degree $> (2k+1)$ in G' , then delete this vertex from graph G', decrement the value of k' by 1 and include this vertex in feedback vertex set S'. If there are many vertices of degree $> (2k+1)$, include the maximum degree vertex among these in the feedback vertex set S'. Apply reduction rule 2. Apply this rule until it is applicable.
We will prove that this rule is safe. If a vertex v is involved in $k+1$ cycles , then that vertex will have a degree $> (2k+2)$. If we do not include this vertex in our feedback vertex set, then we must include one vertex from each cycle in the feedback vertex set S'. But feedback vertex set S' should be of size atmost k. So v must be included in feedback vertex set.

Reduction Rule 2 : If any vertex has degree ≤ 1. Then remove these vertices from graph G'. Parameter k' and feedback vertex set S' remains unchanged. If G' has a vertex of degree 0, which means this vertex is an isolated vertex which need not have to be included in feedback vertex set. If G' has a vertex of degree 1, this vertex cannot be involved in any cycle, so removal of this vertex will not affect the graph G'. So, these vertices of degree ≤ 1 can be safely removed.
Now every vertex has degree $\leq (2k+1)$. We may have many connected components at this stage. If the number of connected components at this stage are greater than k', then feedback vertex set must include atleast one edge from each of these components which violates the statement that S' must be of size atmost k'. So, in this case, we return a NO instance. If the number of connected components are less than k', then we can apply next rule for each connected component.

Reduction Rule 3 : If there exists a set of vertices X in G' whose removal from G' will result in spanning tree F' with maximum number of edges. Let x be the maximum degree vertex in F'. Removal of x from F' results in many connected components as x has bridge edges to each connected component in $F' - x$. Now we remove these bridge edges and add double edges from x to every vertex in X.

Our feedback vertex set may contain X as subset because every vertex in X is involved in some or the other cycle, otherwise this vertex could have been added to spanning forest. We remove the edges incident on x in F' as this is forming cycles with every vertex in X. In the end, to compensate with these cycles that are formed by x with each vertex in X, we add double edges from x to every vertex in X. G' is the resulting graph with changes made only to x. There still exist some cycles in G' as otherwise X would be empty if there are no cycles. So, we apply our next rule which is the final rule.

Reduction Rule 4 : We consider $G' - x$ here. Apply reduction rule 2 on this which ensures every vertex in $G' - x$ is involved in atleast one cycle. X is the set which is forming the cycles, as otherwise they could have been added to F'. Let y be a vertex in X. We try to find a path from y to y in G' where the internal vertices in this path belong to $V' - X$. If such a path is found, we add a self loop on y. If

such a path is not found, we move on to next vertex in X. Two vertices from X cannot have the same path to themselves, as otherwise the common point of the two paths would be there in X instead of these two vertices. Once we read all vertices in X, we return our kernel G'' which is equal to $G'(X \cup x)$. There can be atmost k' connected components in G' before applying reduction rule 3. So there can be atmost k' vertices in X and atmost one x for each connected component. So kernel has $\mathcal{O}(2k')$ vertices.

Once we find the kernel, we can run any FPT algorithm on this to find the final feedback vertex set. Final Feedback vertex set is $S = S' \cup$ FPT(kernel) where FPT(kernel) refers to applying FPT algorithm on kernel.

Let us prove that this algorithm is safe.

Theorem 6.3.1. *S is an feedback vertex set of size atmost k for G iff there is a feedback vertex set S' of size atmost k for G'.*

Proof. Let us first prove the forward direction ie., if S is a feedback vertex set of size atmost k for G then there exists a feedback vertex set S' of size atmost k for G'. We shall construct S' which is initially ϕ. For every vertex u in S, there exists two cases :

i If $u \in G'$, then

   a. If u has a self loop, then this vertex can be included in S'.

   b. If it does not have a self loop, then either this vertex can be included in S' or the other vertex which has double edges from u can be included in S'. So atmost one vertex is included any time.

ii If $u \notin G'$, then this can be replaced by atmost one vertex in G'. G' contains only those vertices which can remove all cycles from the original graph. So, if u is not there in G', then any other vertex must be hitting all cycles caused by u. So u can be replaced by any other vertex.

iii In this way, S' has size atmost k.

Now let us prove the backward direction ie., if there is a feedback vertex set S' of size atmost k for G' then S' is a feedback vertex set for G too. G' contains all those vertices that hits all the cycles. So if S' is the feedback vertex set of G', then it must be the feedback vertex set of G. □

6.3.3 Fixed Parameter Tractable algorithm for Feedback Vertex Set

This algorithm is taken from [28].

Algorithm 6.3.2 UFVS_FPT_Algorithm

Input: Kernel G', parameter k'.
Require: Partial FVS S' from kernel. Let $S = \phi$ be the final FVS.
1: **if** G' is acyclic **then**
2:     Answer YES and return ϕ.
3: **end if**
4: **if** $k' = 0$ and G' has a cycle **then**
5:     Return NO and exit.
6: **end if**
7: **if** G' has any vertex with degree ≤ 1 **then**
8:     Remove those vertices.
9: **end if**
10: Find a shortest cycle C in G'.
11: **for** some vertex $z \in C$ **do**
12:     **if** UFVS_FPT_Algorithm($G' - z, k' - 1$) is YES **then**

13: Answer YES.
14: $S = S' \cup z$.
15: Return $S\cup$ UFVS_FPT_Algorithm($G' - z, k' - 1$).
16: **else**
17: Answer NO.
18: **end if**
19: **end for**

6.3.4 Solution 2

We will reduce the problem of finding undirected feedback vertex set to the problem of d-Hitting Set. d-Hitting set is defined as follows :
Input: A set U and a family F of subsets of U each of size at most d, and $k \epsilon N$.
Question: Is there a set H of at most k elements of U that has a nonempty intersection with each set in F?

d - Hitting set with U as vertex set of G , F as the all cycles in G and d as the number of vertices in longest cycles of G is nothing but an instance of undirected feedback vertex set. The algorithm to find feedback vertex set in an undirected graph is as follows :

Algorithm 6.3.3 Undirected Feedback Vertex Set : Solution 2

1: ***Pass 1***:
Input: A stream which consists of edges of a simple undirected unweighted graph $G(V, E)$ in the form $(u, v) \in [n] \times [n]$ where n is known but m is not known and $u < v$. Stream follows arbitrary model.
Input: A parameter k.
Require: Maintain kernel graph $G' = \phi$ and an integer $d = 0$.
2: **for** every edge (u, v) in the stream **do**
3: Store the edge in the graph G'.
4: **if** $|G'| > (n + nk)$ **then**
5: Return a NO instance.
6: **end if**
7: **end for**
8: Run any algorithm to find all cycles in G' and write these cycles to W-Stream which can be used as input to next pass. Let d be the length of largest cycle.
9: ***Pass 2***:
Input: W-Stream from previous pass.
Input: d which is the length of largest cycle.
10: This becomes an instance of d-hitting set whose family of subsets are cycles from the W-Stream. The output of d-hitting set is a set of vertices that hits all subsets(ie cycles) which is what we want.

We can refer to [19] for finding streaming kernel of d-Hitting Set.

Chapter 7

Conclusion and Future Work

We present algorithms for finding connectivity in directed acyclic dynamic graph. We also proposed a solution for the open problem on matchings in turnstile streams. We also presented algorithms for finding kernels of various graph problems including edge dominating set and undirected feedback vertex set in parameterized streaming model. Our algorithm for edge dominating set works for both insert-only and insert-delete model whereas algorithm for undirected feedback vertex set works in insert-only model.

There are many possible directions for future research. Naturally, it would be interesting to improve existing results like reducing the space used for all the problems proposed in this thesis and improving the approximation ratio when estimating matching size and finding connectivity in generalized directed graphs. Some other problems include considering other NP-Hard problems in parameterized streaming framework. Other specific questions can be found at the wiki, Sublinear Open Problems.

Bibliography

[1] FPT and treewidth. `http://courses.csail.mit.edu/6.854/06/scribe/s21-fixedParamNP.pdf`.

[2] Charu C Aggarwal. *Data streams: models and algorithms*, volume 31. Springer Science & Business Media, 2007.

[3] Kook Jin Ahn, Sudipto Guha, and Andrew McGregor. Analyzing graph structure via linear measurements. In *Proceedings of the twenty-third annual ACM-SIAM symposium on Discrete Algorithms*, pages 459–467. SIAM, 2012.

[4] Kook Jin Ahn, Sudipto Guha, and Andrew McGregor. Graph sketches: sparsification, spanners, and subgraphs. In *Proceedings of the 31st symposium on Principles of Database Systems*, pages 5–14. ACM, 2012.

[5] Surender Baswana, Manoj Gupta, and Sandeep Sen. Fully dynamic maximal matching in $\mathcal{O}(\log n)$ update time. In *Proceedings-Annual Symposium on Foundations of Computer Science*, pages 383–392. IEEE, 2011.

[6] Hans L Bodlaender. A cubic kernel for feedback vertex set. In *STACS 2007*, pages 320–331. Springer, 2007.

[7] Kevin Burrage, Vladimir Estivill-Castro, Michael Fellows, Michael Langston, Shev Mac, and Frances Rosamond. The undirected feedback vertex set problem has a poly (k) kernel. In *Parameterized and exact computation*, pages 192–202. Springer, 2006.

[8] Amit Chakrabarti. Data stream algorithms. *Lecture Notes, Dartmouth College*, 2009.

[9] Rajesh Chitnis, Graham Cormode, Hossein Esfandiari, MohammadTaghi Hajiaghayi, Andrew McGregor, Morteza Monemizadeh, and Sofya Vorotnikova. Kernelization via sampling with applications to dynamic graph streams. *arXiv preprint arXiv:1505.01731*, 2015.

[10] Rajesh Chitnis, Graham Cormode, MohammadTaghi Hajiaghayi, and Morteza Monemizadeh. Parameterized streaming algorithms for vertex cover. *arXiv preprint arXiv:1405.0093*, 2014.

[11] Rajesh Chitnis, Graham Cormode, MohammadTaghi Hajiaghayi, and Morteza Monemizadeh. Parameterized streaming: maximal matching and vertex cover. In *Proceedings of the Twenty-Sixth Annual ACM-SIAM Symposium on Discrete Algorithms*, pages 1234–1251. SIAM, 2015.

[12] Rajesh Chitnis, Marek Cygan, Mohammataghi Hajiaghayi, and Dániel Marx. Directed subset feedback vertex set is fixed-parameter tractable. *ACM Transactions on Algorithms (TALG)*, 11(4):28, 2015.

[13] Michael S Crouch, Andrew McGregor, and Daniel Stubbs. Dynamic graphs in the sliding-window model. In *Algorithms–ESA 2013*, pages 337–348. Springer, 2013.

[14] Marek Cygan, Marcin Pilipczuk, Michal Pilipczuk, and Jakub Onufry Wojtaszczyk. Subset feedback vertex set is fixed-parameter tractable. *SIAM Journal on Discrete Mathematics*, 27(1):290–309, 2013.

[15] Camil Demetrescu, David Eppstein, Zvi Galil, and Giuseppe F Italiano. Dynamic graph algorithms. In *Algorithms and theory of computation handbook*, pages 9–9. Chapman & Hall/CRC, 2010.

[16] Rod Downey. A parameterized complexity tutorial. In *Language and Automata Theory and Applications*, pages 38–56. Springer, 2012.

[17] Rod G Downey and Michael R Fellows. Fixed-parameter tractability and completeness i: Basic results. *SIAM Journal on Computing*, 24(4):873–921, 1995.

[18] Rodney G Downey and Michael R Fellows. *Fundamentals of parameterized complexity*, volume 4. Springer, 2013.

[19] Stefan Fafianie and Stefan Kratsch. Streaming kernelization. In *Mathematical Foundations of Computer Science 2014*, pages 275–286. Springer, 2014.

[20] Wai Shing Fung, Ramesh Hariharan, Nicholas JA Harvey, and Debmalya Panigrahi. A general framework for graph sparsification. In *Proceedings of the forty-third annual ACM symposium on Theory of computing*, pages 71–80. ACM, 2011.

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

[22] Marina Horlescu. Graph sketches. 2013.

[23] Konstantin Kutzkov and Rasmus Pagh. Triangle counting in dynamic graph streams. In *Algorithm Theory–SWAT 2014*, pages 306–318. Springer, 2014.

[24] Silvio Lattanzi, Benjamin Moseley, Siddharth Suri, and Sergei Vassilvitskii. Filtering: a method for solving graph problems in mapreduce. In *Proceedings of the twenty-third annual ACM symposium on Parallelism in algorithms and architectures*, pages 85–94. ACM, 2011.

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

[26] Shanmugavelayutham Muthukrishnan. *Data streams: Algorithms and applications*. Now Publishers Inc, 2005.

[27] Monika R Henzinger Prabhakar Raghavan. Computing on data streams. In *External Memory Algorithms: DIMACS Workshop External Memory and Visualization, May 20-22, 1998*, volume 50, page 107. American Mathematical Soc., 1999.

[28] Venkatesh Raman, Saket Saurabh, and CR Subramanian. Faster fixed parameter tractable algorithms for finding feedback vertex sets. *ACM Transactions on Algorithms (TALG)*, 2(3):403–415, 2006.

[29] Stéphan Thomassé. A $4k^2$ kernel for feedback vertex set. *ACM Transactions on Algorithms (TALG)*, 6(2):32, 2010.

[30] Mingyu Xiao, Ton Kloks, and Sheung-Hung Poon. New parameterized algorithms for the edge dominating set problem. In *Mathematical Foundations of Computer Science 2011*, pages 604–615. Springer, 2011.

[31] Mariano Zelke. Optimal per-edge processing times in the semi-streaming model. *Information Processing Letters*, 104(3):106–112, 2007.

[32] Jian Zhang. A survey on streaming algorithms for massive graphs. In *Managing and Mining Graph Data*, pages 393–420. Springer, 2010.