1 Introduction

We are now turning to finding connected components in the semi-streaming setting. We will see two algorithms, each for a specific setup in the streaming model.

2 Preliminaries

2.1 Notation

We will use the notation $\tilde{O}(f)$ to hide poly-logarithmic factors in $f$, i.e., $\tilde{O}(f) = O(f \cdot \text{poly log } f)$.

By saying that an event $E$ happens with high probability (whp), we refer that $\Pr [E] \geq 1 - n^{-c}$ for some constant $c \geq 1$.

2.2 Streaming setting

An algorithm $A$ for a problem $P$ is said to be a streaming algorithm if:

- $A$ scans the input of size $N$ element by element, e.g., edge by edge of a graph or entry by the entry of an array. We also say that element by element arrive.

- $A$ outputs a solution to $P$ while at any moment its memory consumption is $\text{poly log } N$ bits.

There are studies of streaming algorithms concerning how many scans/passes over the input $A$ can make. In this class, we assume that $A$ makes only one pass/scan over the inputs.

2.3 Semi-streaming setting

Many problems are incredibly challenging in the streaming setting. Moreover, output for many graph problems cannot be stored in $\text{poly log } n$ memory. That inspired several researchers to formalize a semi-streaming setting $[FKM+05]$. In this setting, if we solve a problem on an $n$-vertex graph, the algorithm can use $O(n \text{ poly log } n)$ bits of memory.

3 Connected components in the semi-streaming setting

A reasonably simple algorithm maintains the connected components for the so-far-seen edges. It is outlined as Algorithm 1.
We can think of a streaming setting as a set of updates on our object. For instance, when we say that there is a stream of edges \((e_1, \ldots, e_m)\), we can also think that we begin with an empty graph. Then, the edge \(e_1\) is added, the edge \(e_2\), and so on. In this case, the edges are also inserted. However, we can also think of settings where edges are deleted. An example of such a stream is \((e_1, 1), (e_2, 1), (e_3, -1), \ldots\), where \((e_1, 1), (e_2, 1)\) means that \(e_1\) and \(e_2\) are inserted, while \((e_3, -1)\) means that \(e_3\) is removed.

### 4.1 \(L_0\) sampler

In Lecture 7, we saw an algorithm that samples an element from a stream in which only insertions are allowed. Perhaps surprisingly, there exists a streaming algorithm that samples an element even in the turnstile setting. We call these algorithms \(L_0\) samplers. Given a vector \(x \in \mathbb{R}^d\), let \(\text{supp}(x)\) be
the number of non-zero coordinates of $x$. It will be convenient to think of $L_0$ sampling as outputting an index in supp$(x)$ such that the following holds

$$
\Pr \left[ \text{output equals } j \right] = \begin{cases} 
\frac{1}{\text{supp}(x)} & \text{if } x_j \neq 0 \\
0 & \text{otherwise}
\end{cases}
$$

**Theorem 1** ([JST11], Section 2.1). Let $x \in \mathbb{R}^{\text{poly} \ n}$ be a non-zero vector. For the turnstile setting, there exists an $L_0$ sampler algorithm that uses $O(\text{poly log } n)$ bits and, with high probability, outputs a coordinate $j \in \text{supp}(x)$.

It is important to note that Theorem 1 does not require $x$ to be stored in the memory; instead, its entries are updated in the streaming fashion. The memory the stated $L_0$ uses includes all the information it stores about $x$.

A handy feature of some of the existing $L_0$ samplers, like the one described in [JST11], is that the algorithm performs a set of linear measurements of $x$, and then uses those measurements to output the desired index. This technique has had a significant impact on designing streaming algorithms. By a linear measurement, we typically refer to a matrix $M \in \mathbb{R}^{k \times d}$ and the fact that the algorithm computes $Mx$; typically, $k \in O(\text{poly log } n)$. $S_x = Mx$ is called a linear sketch. Linear sketches are so powerful as they are additive. In particular,

$$
S_x + S_y = Mx + My = M(x + y).
$$

In other words, taking the sum of linear sketches computed on two distinct vectors is the same as computing a linear sketch on the sum of those two vectors. We will next see how those properties can be highly beneficial in solving problems.

### 4.2 Connected components

We now phrase the problem of computing connected components in the language that fits well into applying the $L_0$ sampler results we mentioned. As the first step, we talk about graph representation. As the second step, we describe a turnstile semi-streaming algorithm that whp outputs connected components.

#### 4.2.1 Graph representation

Our algorithm constructs several $L_0$ samples for the neighborhood of each vertex. To apply Theorem 1, it is convenient to think about the neighbors of a vertex as a vector. To that end, for each vertex $v$, we define vector $x^v \in \mathbb{R}^n$ as follows

$$
\begin{align*}
x^v_{(a,b)} &= \begin{cases} 
1 & \text{if } v = a < b \text{ and } \{a, b\} \in E \\
-1 & \text{if } a < b = v \text{ and } \{a, b\} \in E \\
0 & \text{otherwise}
\end{cases}
\end{align*}
$$

One might wonder why some entries in $x^v$ are positive while some negative. This representation has a very convenient property. Namely, consider any subset of vertices $U \subseteq V$. Then, supp$(\sum_{w \in U} x^w)$ contains the edges incident to $U$ but excluding the edges within $U$. That is, for two neighboring vertices $a$ and $b$ such that $a < b$, we have that $(x^a + x^b)_{(a,b)} = 0$. We build on these observations to design our algorithm.
4.2.2 Algorithm

We are almost ready to provide our algorithm. Before that, we mention that the matrix $M$ discussed below Theorem 1 depends on a set of random bits $r$. So, to emphasize that it is a function of a string of random bits, instead of $L_0$ sampler only, we write $L_0^r$ sampler in our algorithm.

Algorithm 2 is inspired by the following simple process for computing CC. Initially, each vertex is a singleton component. Then, for $\log n$ steps, we perform the following. Each current CC $C$ samples an edge $C_e$ that connects $C$ to another CC; this sampling is done simultaneously for all the components. Then, each component $C$ – one after another – merges with the component connected to it by $C_e$. Algorithm 2 uses $L_0^r$ sampling to implement this idea.

**Input**: A graph $G = (V, E)$ given as a stream of edges. Assume that the vertices in $V$ are labeled 1 through $n$.

```plaintext
/* Initialization. */
1 Fix $r_1, \ldots, r_{\log n}$ random-bit strings that will be used for computing $L_0^r$ samplers.
2 For each vertex $v$ and each random-bit string $r_i$, we will maintain an $L_0^r(v)$ sampler for $x^v$.

/* Stream processing. */
3 for edge $(e = \{u, v\}, \text{FLAG})$ on the stream do
   // If FLAG equals true, then $e$ is added to the graph, and otherwise $e$ is removed from the graph. An edge is removed only if it was previously added.
   for $i = 1 \ldots \log n$ do
      Update $L_0^r(u)$ and $L_0^r(v)$ based on $(e, \text{FLAG})$.

/* Stream post-processing. */
6 Mark each vertex as a singleton CC.
7 for $i = 1 \ldots \log n$ do
   8 For each current connected component $C$ obtain an $L_0^r(C)$ sampler for $\text{supp}\left(\sum_{w \in C} x^w\right)$ by combining the linear sketches of $L_0^r(w)$ for all $w \in C$.
   9 If such an edge exists, using $L_0^r(C)$ sample an edge $e$ incident to $C$. Merge the two connected components incident to $e$.
10 return the computed connected components
```

Algorithm 2: A turnstile algorithm for finding connected components.

Success probability. To analyze the algorithm, we first recall that each of the $L_0$ samplers provides correct output with high probability, i.e., with probability at least $1 - n^{-c}$ for any arbitrary large constant $c$. So, by the union bound over all the samplers, we have that they provide correct output whp.

Number of samplers. Why did we use $O(\log n)$ $L_0$ samplers instead of only 1 per vertex? The reason is that once we use a sampler $L_0^r(v)$ to sample an edge incident to a component $C$ that contains vertex $v$, we have revealed a part of $L_0^r(v)$’s randomness, so it is not clear that $L_0^r(v)$ can be used again to provide the desired output.

Number of iterations. Why $\log n$ iterations on Line 7 suffice? Because each time we perform an iteration, the smallest non-maximal CC at least doubles in size.

Memory complexity. Since each vertex maintains $O(\log n)$ samplers, and each sampler uses $\text{poly log } n$ bits of memory, the total memory consumption of Algorithm 2 is $O(n \text{poly log } n)$ bits.
References
