CS 38 Introduction to Algorithms **Week 8 Recitation Notes**

Streaming, Network Flow

TA: Joey Hong (jhhong@caltech.edu)

1 Streaming Algorithms

We can imagine a situation in which a stream of data is being recieved but there is too much data coming in to store all of it. However, we want to extract some information out of the stream of data without storing all of it. An example could be a company like Facebook or Google storing click data or login data. What they really care about is aggregate data and not individual datum.

The following are all examples of streaming problems that we might be interested in. Some you will find have very simple algorithms and some are rather complicated. As an aside, it should be noted that the majority of work in streaming algorithms requires using randomized algorithms (which you can learn all about in CS/CMS 139). Typically, this is done by selecting a small sample of the data and estimating the aggregate data from the small sample. [1]

- Calculating the length of the stream.
- Calculating the mean or median of a set of values.
- Calculating the standard deviation of a set of values.
- Calculating the number of unique elements (assuming the stream is elements from $\{1, ..., n\}$). [2]
- Calculating the k most frequent elements. [3]
- Sampling elements according to the distribution of the stream.

1.1 Basic Setup

Definition (Stream). A sequence σ of m elements $(s_1, .., s_m)$ (called *tokens*), from a universe of size n, given by $[n] := \{1, 2, ..n\}$. Typically, the length of the stream is unknown and n is large, but $n << m$.

Streaming algorithms compute some function of an input stream σ. The goal of streaming algorithms will be the process σ using a "small" amount of memory s; specifically, we want s to be *sublinear* in the input size, $o(m)$ (otherwise, we can naively store the entire sequence in $O(m)$).

The "ideal" streaming algorithm uses space $s = O(\log n + \log m)$. You notice that this is the amount of space required to store a constant number of stream elements, and a constant number of counters that iterate up to the length of σ.

A streaming algorithm should also access the input stream sequentially (hence, stream), as opposed to random access of the tokens.

[1] I literally copy-pasted from Chapter 9 of TA Notes
[2] Also called 0-th moment. The k-th moment is $F_k(\sigma) = \sum_{i=1}^{m} s_i^k$.
[3] Also called heavy-hitters problem.

1.2 Sampling

Reservoir sampling is a family of randomized algorithms, whose goal is to sample k elements from a list of size m, following some probability distribution.

Below is an example of a question that has been popular in data scientist interviews. [4] It is essentially reservoir sampling with $k = 1$:

Problem (Uniform Sampling). Given a large stream σ, create a streaming algorithm that randomly chooses an item from this stream such that each is equally likely to be selected.

Notice that for our algorithm to work, on processing the last element s_m, we must select s_m with $1/m$ probability. As this is our only choice, we extend this approach for all elements s_i.

Algorithm (Uniform Sampling).

- Initialize: Set $x \to$ null.
- Process s_i: On seeing the i-th element, $x \to s_i$ with probability $1/i$.
- Output: x.

Analysis. What is the probability that $x = s_i$ after all m elements are processed?

$$\begin{aligned}
P(x = s_i) &= P(s_i\text{was chosen when being processed}) \prod_{j>i} P(s_j\text{was not chosen when being processed}) \\
&= \frac{1}{i} \times (1 - \frac{1}{i+1}) \times \ldots \times (1 - \frac{1}{m}) \\
&= \frac{1}{i} \times \frac{i}{i+1} \times \ldots \times \frac{m-1}{m} \\
&= \frac{1}{m}
\end{aligned}$$

It is easy to see correctness of the algorithm from here.

To get a single sample, we only need $O(\log n + \log m)$ bits of space ($O(\log n)$ from storing x, and $O(\log m)$ from keeping track of the current index of element in stream). This is really good.

1.3 Counting Distinct Elements

Problem (Count-Distinct). Given a large stream σ with $n << m$, find an estimate $\hat{n}$ of n. [5]

There are many approaches to tackling this problem, many of which involve hash functions. All such algorithms will be randomized, and we will attempt to see why that must be the case.

Lower Bound on Memory for Exact Algorithm. We argue that any deterministic algorithm that solves Count-Distinct requires $O(n)$ bits of memory. To see this, we first notice that the set of distinct elements is one of 2^n subsets of $\{1, 2, ..., n\}$. If we wanted a different memory state for each possible subset, that gives us n bits are required.

Now, we need to prove that if any two data streams result in the same memory state, our algorithm would give the incorrect output for at least one of them. Suppose first that we have seen $k < m$ elements

[4] I cannot confirm if that's actually true

[5] Cover only if time permits

of the stream. Now, assume that there are two input streams σ_1, σ_2 that result in the exact same memory state at the end of the algorithm. There are two cases: (1) σ_1 and σ_2 have different numbers of distinct elements, or (2) same number of distinct elements.

Consider the first case. Obviously, our algorithm would produce the same number of distinct symbols for both input streams, and will be wrong on at least one of them. Now, consider the second case. Upon seeing the next symbol in σ_1 that is not in σ_2, our algorithm will either change its memory state to include that symbol, or ignore it, and both cases result in an incorrect output for at least one of the streams.

□

Alternate Proof:[6]

Suppose you had a deterministic algorithm that solves the problem in space S. For $x \in \{0,1\}^n$ consider the stream $((1, x_1), .., (n, x_n))$. This stream has n distinct elements.

Suppose we execute the algorithm; at the end of the stream the state of the algorithm's memory is some arbitrary string $m(x) \in \{0,1\}^S$. Now, let's initialize the algorithm's memory to $m(x)$ (for some x) and give it a single element $(i, 0)$. If the number of distinct elements reported by the algorithm increases by 1 then we know that $x_i = 1$, whereas if it doesn't change we know that $x_i = 0$.

In this way it is possible to completely recover x from $m(x)$, therefore $m(x)$ must take at least 2^n possible values: $S \geq n$.

□

Hopefully, this kind of argument gives you some intuition on when to resort to randomized algorithms. Now, we will try to shoot for $O(\log n)$ using a randomized algorithm. We will try to solve this problem without the context of hash functions. To do so, we require a much stronger assumption about the data stream; let $S \subset [n]$ be the set of distinct elements in σ, we assume that elements of S are sampled uniformly from $[n] = \{1, 2, ..., n\}$.

Algorithm (Count-Distinct).

- Initialize: Set $min \to$ null.
- Process s_i: On seeing the i-th element, set $min \to \min(min, s_i)$.
- Output: $\frac{n}{min} - 1$.

Analysis. If the elements of S are sampled uniformly from $\{1, 2, .., n\}$, then the elements partition the set $\{1, 2, .., n\}$ into $|S| + 1$ subsets of size approximately $\frac{n}{|S|+1}$.

Thus, the minimum element of S should have value kind of close to $min = \frac{n}{|S|+1}$. Solving for $|S|$ gives $|S| = \frac{n}{min} - 1$. This is a very bad estimate, but we can at least prove some bounds on it.

Lemma. Let $S = \{s_1, s_2, .., s_d\}$, and $|S| = d$ be the correct answer. For any k. with probability $\geq 1 - (\frac{1}{k})^2$, we have $\frac{d}{k} \leq \frac{n}{min} \leq kd$.

Proof. First, we will show that $P(\frac{n}{min} > kd) \leq \frac{1}{k}$.

$$P\left(\frac{n}{min} > kd\right) = P\left(min < \frac{n}{kd}\right) = P\left(\exists i,\, s_i < \frac{n}{kd}\right)$$

From the Union Bound [7], we get the inequality,

$$P\left(\exists i,\, s_i < \frac{n}{kd}\right) \leq \sum_{i=1}^{d} P\left(s_i < \frac{n}{kd}\right)$$

[6] Credit to something Chinmay found

[7] Let X_i be the event that $s_i < \frac{n}{kd}$. The Union Bound states $P(\cup_i X_i) \leq \sum_i P(X_i)$

Since we assume s_i are sampled uniformly from $\{1, 2, ..., n\}$, we have $P(s_i < \frac{n}{kd}) = 1/kd$,

$$P\left(\frac{n}{min} > kd\right) \leq \sum_{i=1}^{d} P\left(s_i < \frac{n}{kd}\right) = d\left(\frac{1}{kd}\right) = \frac{1}{k}$$

Next, we show that $P(\frac{n}{min} < \frac{d}{k}) \leq \frac{1}{k}$.

$$P\left(\frac{n}{min} > \frac{d}{k}\right) = P\left(min > \frac{kn}{d}\right) = P\left(\forall i,\, s_i > \frac{kn}{d}\right)$$

Unfortunately, no simple identities exist to bound this result, so we will have to be a little clever. Feel free to skip this if you do not have a solid foundation in probability, though I will remain on the order of Ma 3. We will define an indicator variable z_i for $i = 1, 2, ..., d$,

$$z_i = \begin{cases} 0 \text{ if } s_i > \frac{kn}{d} \\ 1 \text{ otherwise} \end{cases}$$

It is easy to see that $E(z_i) = \frac{k}{d}$, and if we let $z = \sum_{i=1}^{d} z_i$, the $E(z) = k$. We can also bound the variances,

$$Var(z_i) = E(z_i^2) - (E(z_i))^2 \leq E(z_i)$$

Using Chebychev's Inequality [8], we have,

$$\begin{aligned} P\left(\forall i,\, s_i > \frac{kn}{d}\right) &= P(y = 0) \leq P(|y - E(y)| \geq E(y)) \\ &\leq \frac{Var(y)}{E^2(y)} \leq \frac{1}{E(y)} = \frac{1}{k} \end{aligned}$$

Combining the two bounds yields the desired inequality. □

The bounds for the algorithm are fairly weak. It basically only gives that $\hat{n}$ is on the same order of magnitude of n, and is not an arbitrarily good estimation. Also, our assumption about a uniformly sampled S is a very strong assumption. We can get rid of it and achieve same error bounds by defining a hash function $h : \{1, 2, .., n\} \rightarrow \{1, 2, ..., k\}$, and running our algorithm on $(h(s_1), .., h(s_m))$. The hash function has to satisfy some properties, but this is not a topic for today.

2 Network Flow

2.1 Problem Definition

The maximum flow problem is a classic in the design and analysis of algorithms. All problems are done on a *capacitated network*, which has the following ingredients:

- A directed graph G, with vertices V and edges E
- A source vertex $s \in V$
- A sink vertex $t \in V$
- A nonnegative capacity function for each edge $C : E \rightarrow \mathbb{R}_+$.

[8] States, for some random variable X with mean μ and standard deviation σ, that $P(|X - \mu| \geq k\sigma) \leq \frac{1}{k^2}$

Formally, a flow is also some function $f : E \to \mathbb{R}_+$. As the output to our problem, we can consider it a nonnegative vector $\{f_e\}_{e \in E}$, indexed by edges, and satisfying the constraints:

- **Capacity constraints:** $f_e \leq C(e)$ for every $e \in E$.
- **Conservation constraints:** For every vertex $v \in V - \{s, t\}$,

$$\text{amount of flow entering } v = \text{amount of flow exiting } v$$

The objective is the compute the *maximum flow*, the flow of edges leaving s with the maximum total value.

$$U(f) = \sum_u f_{(s,u)}$$

2.2 Augmenting Path Method

We can convert a capacitated graph $G = (V, E, C, s, t)$ with flow f into a residual graph $G_f = (V, E_f, s, t)$. To do so, every edge $e \in G$ is converted to a forward edge with weight $C(e) - f_e$, and a backwards edge with weight f_e. Thus, there are at most two edges for each edge in G.

Given the residual graph G_f, we can solve the flow problem in G by considering augmenting paths (s-t) on G_f, and pushing flow along those paths. The pseudocode for the algorithm is described below:

Algorithm (Ford-Fulkerson).

```
• Initialize $f_e = 0$ for all $e \in E$.
• Repeat until no augmenting paths remain:
    // Function FindPath($G_f$) finds $s$-$t$ path in the residual graph $G_f$.
    P = FindPath($G_f$)
    Let $\Delta = \min_{e \in P}$($e$'s weight in $G_f$).
    Augment $f_e$ on all edges $e \in P$ by $\Delta$.
• Return $f$ as the maximum flow.
```

Lemma (Optimality of Augmenting Paths). If f is a flow in G such that the residual network G_f has no s-t path, then the f is a maximum flow of G. [9]

We can vary the FindPath method to be any graph search algorithm. Edmonds-Karp is a special case of Ford-Fulkerson where they find the shortest augmenting path in G_f using BFS. Doing such a search takes $O(m)$ time, and there are at most $O(mn)$ path augmentations [10], so Edmonds-Karp has a runtime of $O(m^2 n)$.

[9] See lecture notes for proof.

[10] Because each path augmentation removes some edge in G_f, so there can be at most m augmentations of a certain path length. There are $O(n)$ total possible path lengths.

2.3 Minimum Cut

The input to the minimum cut problem is the same as that of maximum flow. Instead, the feasible solutions are s-t cuts, meaning partitions of the vertices V into A, B such that $s \in A$, $t \in B$.

The capacity of a cut is the total capacity of edges coming out of A (not going into), and can be written formally as

$$C(A, B) = \sum_{e \in \delta^+(A)} C(e)$$

Theorem (Max-Flow Min-Cut). The minimum s-t cut is equal to the maximum flow. Formally,

$$\max_f U(f) = \min_S C(S, V - S)$$

From the lectures, you know that the max-flow min-cut theorem is actually a case of strong duality. You will learn in less than one week why they are dual problems. For now, we just know that maximum flow is equal to the minimum cut, but how do we compute the cut? The answer is again in the residual graph

Algorithm (Min-Cut).

- Run Ford-Fulkerson and consider the residual graph G' at output.
- Mark $A \leftarrow$ all vertices reachable from s in G'.
- Return $S = \{(u, v) \in E \,|\, u \in A, v \in V - A\}$.

It is pretty easy to see that S is a valid cut, and has size equal to the value of the maximum flow.

2.4 Bipartite Matching

One of the most important applications of maximum flow is in maximum bipartite matching. As you probably already know, a bipartite graph is a graph where its vertices can be split into two sets such that there is not edge between vertices from the same set. They are often denoted $G = (V \cup W, E)$

A matching in a graph is a set of edges $S \subset E$ such that no vertex is touched by more than one edge (i.e. edges share no endpoints). A maximum matching is just the matching with maximum cardinality. A perfect matching matches all of the vertices.

Proposition. Maximum bipartite matching reduces in linear time to maximum flow.

Proof. Given an undirected bipartite graph $G = (V \cup W, E)$, we construct a directed graph $G' = (V', E')$ as follows:

(1) Add source and sink, so $V' = V \cup W \cup s, t$.

(2) Direct all edges $e \in E$ from V to W, and add directed edges from s to every vertex in V, then from every vertex in W to t.

(3) Give all edges incident to s or t capacity 1. Give all other edges infinite capacity.

Claim: The maximum flow in G' is the maximum bipartite matching in G.

Given a matching of cardinality k in G, we will prove it is the value of a corresponding flow in G'. Consider a flow f that sends 1 unit along each of paths corresponding to the k edges in the matching. It is easy to check that f is a valid flow, with value k.

Now, given a flow f in G' of value k, we will prove it is the value of the corresponding maximum matching. We see that k must be integral, since flows alongs paths are all 0-1 (convince yourself why). Now, consider the set M of edges from V to W in G' with $f(e) = 1$. Each vertex in V and W are only part of at most one edge in M, which forms a matching of size k.

So, we have a bijective map, where every edge $(v, w) \in M$ a matching of G corresponds to one unit of flow $s \to v \to w \to t$ in G'.

$\square$

The transformation is in $O(m+n)$, which is linear in the input. [11]

2.5 Problems

The difficulty in max-flow problems is often in recognizing that it is one. Very rarely will the problem deliberately mention transporting elements across roads or pipes [12]. In the following problems, this difficulty doesn't exist (as you can be 99% sure I will only include flow-related problems). Still, it is sometimes nontrivial how a capacitated graph is constructed.

1. You and your $k-1$ friends live in a county comprised of N cities connected via some one-way roads. You guys all want to get from city a to city b; however, all of your vehicles are so bad that they destroy any road that they take, making them unusable in the future. Give an algorithm that determines whether it is possible for all k of your to make it to city b.

2. We are given a $n \times m$ chessboard, with k squares cut out of the board. You want to know the maximum number of rooks that can be placed on this chessboard, such that no two rooks are in position to attack eachother.

Rooks cannot be placed on the k cut squares, but otherwise, a rook can still attack anything in its row or column, regardless of whether there is a cut out square on its path.

Devise an algorithm to determine the maximum number of rooks to be placed on this chessboard.

[11] A method of solving maximum bipartite matching directly is via Hopcroft-Karp, which also deals with augmenting paths. That algorithm runs in $O(m\sqrt{n})$

[12] A goal runtime of $O(N^3)$ is pretty common for max-flow problems