Error Correcting Codes: Combinatorics, Algorithms and Applications (Spring 2010)

Lecture 40: Hot Items Problem

April 21, 2010

Lecturer: Atri Rudra *Scribe: Shujie Liu*

Last lecture we talked about solving the problem of list recovery with Guruswami-Sudan algorithm and looked at the requirements of data stream algorithms via the example of the hot items problem. In today's lecture, we will explore the connection between hot items problem and group testing. In particular, the existing strongly explicit construction of d-disjunct matrix based on Reed-Solomon codes will be used to solve this problem.

1 Data Stream Algorithms

We start with the requirements imposed by data stream algorithms and the definition of the hot items problem. After this, we will look into possible solutions for this problem.

Definition 1.1. *A data stream algorithm has four requirements listed below:*

1. *The algorithm should make one pass over the input.*
2. *The algorithm should use poly-log space. (In particular, it cannot store the entire input.)*
3. *The algorithm should have poly-log update time.*
4. *The algorithm should have poly-log reporting time.*

Definition 1.2. ***(Hot Items Problem)*** *Given n different items, for m input pairs of data $(i_\ell, u_\ell)(1 \leq \ell \leq m)$, where $i_\ell \in [n]$ indicates the item index and u_ℓ indicates corresponding count. The problem requires update the count $f_\ell(1 \leq \ell \leq m)$ for each item, and output all item indices j such that $f_j > \frac{\sum_{\ell=1}^{n} u_\ell}{d}$.*

In this lecture, we will think of d as $O(\log n)$. Hot items problem is also called heavy hitters problems. We state the result below without proof:

Theorem 1.3. *Computing hot items exactly by a deterministic one pass algorithm needs $\Omega(n)$ space (even with exponential time).*

This theorem means that, we cannot solve the hot items problem in poly-log space as we want. However, we could try to find solutions for problems around this. The first one is to output an approximate solution, which will output a set that contains all hot items and some non-hot items.

For this solution, we want to make sure that the size of the output set is not too large (e.g. outputing $[n]$ is not a sensible solution).

Another solution is to make some assumptions on the input. For example, we can assume Zipf-like distribution of the input data, which means only a few items appear frequently. More specifically, we can assume *heavy-tail distribution* on the input data, as:

$$\sum_{\ell:\text{non-hot}} f_\ell \leq \frac{m}{d} \tag{1}$$

This is true for lots applications, such as hot stock finding, where only a few of them have large frequency. Next, we will explore the connection between group testing and hot items problem based on this assumption.

2 Connection to Group Testing

In this part, we will look into the connection between group testing and hot items problem established by [1].

For the hot items problem defined in Definition 1.2, let us first consider a $t \times n$ d-disjunct matrix M, where n is the number of items and $t = O(d^2 \log^2 n)$, as shown in the following.

$$\begin{bmatrix} M_{11} & M_{12} & M_{13} & \cdots & M_{1n} \\ M_{21} & M_{22} & M_{23} & \cdots & M_{2n} \\ \vdots & \vdots & \vdots & \ddots & \vdots \\ M_{i1} & M_{i2} & M_{i3} & \cdots & M_{in} \\ \vdots & \vdots & \vdots & \ddots & \vdots \\ M_{t1} & M_{t2} & M_{t3} & \cdots & M_{tn} \end{bmatrix}$$

Note that we have a strongly explicit construction of M with $t = O(d^2 \log^2 n)$. Define S_i such that $j \in S_i$ if and only if $M_{ij} = 1$. Let C_i be the total count of all $j \in S_i$. What we want to do is to update $C_i (1 \leq i \leq t)$ for each input and report hot items based on C_i and the matrix M. Besides this, we also need to maintain total number of input pairs m. The initialization and update process as well as reporting problem are shown as following:

Initialization:
$m \leftarrow 0$, $C_i \leftarrow 0$, for $1 \leq i \leq t$.

Update:
For each input pair (j, u) with $j \in [n], u \in \mathbb{Z}$:
$m \leftarrow m + 1$, $C_i \leftarrow C_i + u$ if and only if $M_{ij} = 1$.

Reporting hot items:

Given current $C_i(1 \leq i \leq t)$ and m, output all hot items indices j.

The problem of reporting hot items turns out to be the decoding problem of group testing. Consider the following two observations:

Observation 2.1. *If $f_j > \frac{m}{d}$ and $M_{ij} = 1$, then $C_i > \frac{m}{d}$.*
That is because $C_i = \sum_{k:M_{ik}=1} f_k \geq f_j$.

Observation 2.2. *For any $1 \leq i \leq t$, if all j with $M_{ij} = 1$ are not hot items, then we have $C_i \leq \frac{m}{d}$.*
This is based on the heavy tail distribution assumption 1 and $C_i = \sum_{k:M_{ik}=1} f_k$.

Therefore, if we define $\mathbf{x} = (x_1, x_2, \ldots, x_n) \in \{0,1\}^n$ with $x_j = 1$ iff j is a hot item, and $\mathbf{r} = (r_1, r_2, \ldots, r_t) \in \{0,1\}^t$ with $r_i = 1$ iff $C_i > \frac{m}{d}$, we will have $r_i = \vee_{j \in S_i} x_j$. The latter claim follows from Observation 2.1 and 2.2 above. This means we have:

$$M\mathbf{x} = \mathbf{r} \tag{2}$$

Note that by definition, $|\mathbf{x}| < d$. Thus reporting the hot items is the same as decoding $\mathbf{x}$ given M and $\mathbf{r}$, which successfully changes the hot items problem into group testing problem. In the following part, we will design and analyze the algorithm above and check if the conditins in Definition 1.1 are met.

3 Analysis of the Algorithm

In this part, we will review the requiements on data stream algorithm one by one and check if the algorithm for the hot items problem based on group testing satisfies them. In particular, we will need to pick M and the decoding algorithm.

1. *One-pass requirement*
   If we use non-adaptive group testing, the algorithm for the hot items problem above can be implemented in one pass, which means each input is visited only once. (If adaptive group testing is used, the algorithm is no longer one pass, therefore we choose non-adaptive group testing.)

2. *Poly-log space requirement*
   In the algorithm, we have to maintain the counters C_i and m. The maximum value for them is mn, thus we can represent each counter in $O(\log n + \log m)$bits. This means we need $O((\log n + \log m)t)$bits to maintain the counters. Given $t = O(d^2 \log^2 n)$ and $d = O(\log n)$, the total space we need to maintain the counters is $O(d^2 \log^2 n(\log n + \log m)) = O(\log^4 n(\log n + \log m)$.

   On the other hand, if we need to store the matrix M, we will need $\Omega(nt)$ space. Therefore, poly-log space requirement can be achieved only if matrix M is not stored directly.

3. *Poly-log update time*
   As shown in the previous part, we cannot store the matrix M directly in order to have poly-log space. Since RS code is strongly explicit, the d-disjunct matrix M is strongly explicit. In the following, we will briefly prove that the update time is $O(t \times \text{poly}\log t)$ if M is strongly explicit.

   Recall that we use a concatenated code for constructing M: $C^* = C_{out} \circ C_{in}$, where C_{out} is a $[k, q]_q$ RS code and C_{in} is an identity matrix:

   $$C_{in} : [q] \to \{0,1\}^q \text{ where } j \to (0, \cdots, 1, \cdots, 0) \text{ with } 1 \text{ at } j^{th} \text{ position.} \tag{3}$$

   Recall that codewords of C^* are columns of the matrix M, and we have $n = q^k$, $t = q^2$. Every time we want to update the counters C_i for a given input pair (j, u), what we need is to find the corresponding column j in M and update C_i if $j \in S_i$.

   That means, the update problem is like an encoding problem, in which given an input message $\bar{\mathbf{m}} \in {\mathbb{F}_q}^k$ specifying which column we want (where $\bar{\mathbf{m}}$ is the representation of $j \in [n]$ when thought of as an element of ${\mathbb{F}_q}^k$), the ouput is $C_{out}(\bar{\mathbf{m}})$ and it corresponds to the column $M_{\bar{\mathbf{m}}}$.

   On the other hand, for the corresponding column $M_{\bar{\mathbf{m}}}$ in M of a given input message $\bar{\mathbf{m}}$, we can partition this coloum into q chunks, each chunk is of length q. It is noticed that $(C_{out}(\bar{\mathbf{m}}))_{i_1} = i_2$ if and only if the i_1^{th} trunk has 1 on its i_2^{th} position and 0 on other positions (recall the definition of C_{in} in 3), which means $M_{\bar{\mathbf{m}}}(i_1 * q + i_2 - q) = 1$. Therefore, we can compute all i such that $j \in S_i$ by computing $C_{out}(\bar{\mathbf{m}})$, where $M_{\bar{\mathbf{m}}}$ is the j^{th} column in M. Because C_{out} is a linear code, it can be computed in $O(q^2 \times \text{poly}\log q)$ time, which means the update process can be done in $O(q^2 \times \text{poly}\log q)$ time. Since we have $t = q^2$, the update process can be finished with $O(t \times \text{poly}\log t)$ time. [1].

4. *Reporting time*
   The computation of $\mathbf{r}$ require $O(t)$ time while the decoding of $\mathbf{x}$ requires $O(nt)$ time (at least according to what we have seen so far). This means we cannot achieve the poly-log reporting time requirement.

In the next lecture, we will look into this problem and to prove that, we can get a poly-log decoding time without designing a new matrix.

References

[1] Graham Cormode and S. Muthukrishnan, *What's hot and what's not: tracking most frequent items dynamically*, ACM Transactions on Database Systems, Volume 30, Issue 1, March 2005.

[1] This is for the case as RS is strongly explicit. In fact even if explicitly store the generator matrix we will need $O(\log q) = O(\log t)$ extra space which is still poly-log space