Sliding HyperLogLog: Estimating cardinality in a data stream

Yousra Chabchoub, Georges Hébrail

Sliding HyperLogLog: Estimating cardinality in a data stream

Yousra Chabchoub, Georges Hébrail
BILAB
Telecom ParisTech
Paris, France
yousra.chabchoub@telecom-paristech.fr
georges.hebrail@telecom-paristech.fr

***Abstract*—In this paper, a new algorithm estimating the number of active flows in a data stream is proposed. This algorithm adapts the HyperLogLog algorithm of Flajolet et al to the data stream processing by adding a sliding window mechanism. It has the advantage to estimate at any time the number of flows seen over any duration bounded by the length of the sliding window. The estimate is very accurate with a standard error of about $1.04/\sqrt{m}$ (the same as in HyperLogLog algorithm). As the new algorithm answers more flexible queries, it needs an additional memory storage compared to HyerLogLog algorithm. It is proved that this additional memory is at most equal to $5mln(n/m)$ bytes, where n is the real number of flows in the sliding window. For instance, with an additional memory of only $35kB$, a standard error of about 3% can be achieved for a data stream of several million flows. Theoretical results are validated on both real and synthetic traffic.**

***Index Terms*—flow; sliding window; hashing; counting**

I. Introduction

We address in this paper the problem of estimating online the number of distinct elements in a massive data stream. This problem has several interesting applications in the fields of traffic engineering and networks security. In fact, some attacks, such as worm propagation or Denial of Service can be detected by supervising the number of active flows [1]. A flow is a sequence of packets defined by the classical 5 tuple composed of the source and destination addresses, the source and destination port numbers together with the protocol type. A sudden increase in the number of active flows should be quickly identified by the network administrator as it is very likely to be related to an attack. Due to the huge amount of data at a very high bit rate (40 Gb/s in OC-768) in actual internet traffic, scalable algorithms are required. They have to operate fast, using a limited small memory. Under these constraints, it is unrealistic to have an exact counting because it clearly needs a memory size proportional to the number of active flows. Some probabilistic algorithms [2], [4], [6] have been developed to estimate the number of flows with an acceptable standard error of few percents. They only require a sublinear memory and are well adapted to online traffic analysis. Based on hashing functions, probabilistic algorithms are simple to implement and quick enough to could deal with the huge traffic bit rate. Moreover, they treat data in only one pass. To infer the estimate, some of these algorithms rely on bit pattern observables in the binary representation of the hashed values. For instance, in LogLog algorithm [4], Durand and Flajolet are interested in the bit pattern "0^R1". They show that the highest position of the leftmost 1-bit is closely related to the total number of observed flows. A similar bit pattern observable is used in Probabilistic Counting algorithm [2]. A second kind of observables is called order statistics observables. For instance, MinCount algorithm [6] is based on the smallest hashed value.

All these algorithms presented above are designed to perform on a static set of data. It means that they can simply estimate the total number of flows for a given fixed traffic. So they can not be applied to an infinite data stream. Moreover, for many network applications, we are mostly interested in traffic characteristics in the near past (few minutes ago for attacks detection). A natural way to adapt these algorithms to data stream is to use sliding window. Among all the presented algorithms, only the MinCount algorithm has a sliding window version with a detailed analysis (see [3]). The objective is to be able to answer the following query at any time "How may distinct flows have been seen over the last w unit of time?" for any duration w smaller than the time window W.

In this paper, a sliding window adaptation is proposed for the last version of the LogLog algorithm called HyperLogLog [5]. This algorithm requires a memory of only $mln_2ln_2(n/m)$ bits, where n is the real number of flows, to estimate the number of flows with a standard error of about $1.04/\sqrt{m}$. More details about the HyperLogLog algorithm are given in section II. The proposed Sliding HyperLogLog algorithm is presented in section III. In particular we show that we maintain the same accuracy as in the original version (HyperLogLog), with an additional small required memory. In practice, with an additional memory of only $35kB$, a standard error of about 3% can be achieved for a data stream of several million flows. The Sliding HyperLogLog algorithm is tested and validated against synthetic and real traffic in section IV.

II. Related work

The proposed algorithm is an adaptation of the HyperLogLog algorithm to the data stream processing. The objective of the HyperLogLog algorithm is to estimate the cardinality of a given multiset S. A multiset is defined as a set, where an element can be repeated. This algorithm is mainly based on the following pattern "0^R1" in the binary representation of the hashed values. More precisely, after hashing all the elements of the multiset, the highest position of the leftmost 1-bit is denoted by R. Notice that R is an order and duplication insensitive observable. The cardinality of S is then deduced from R via a stochastic averaging process (see Figure 1). This latter mechanism consists in splitting S in m buckets, based on the first b bits in the binary representation of the hashed values, where $m = 2^b$. The objective is to process the buckets independently and to compute an average of R, in order to have a more accurate estimation of the multiset cardinality.

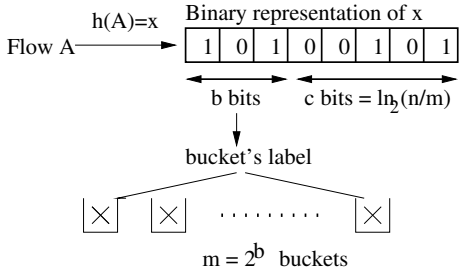

Fig. 1. The stochastic averaging process in the HyperLogLog Algorithm

The pseudo-code of the HyperLogLog algorithm is given below:

Algorithm HyperLogLog ($m = 2^b$ buckets)
Initialize m registers, $R[1], ..., R[m]$ to 0;
For $A \in S$ do

- set $x := h(A)$;
- set $j :=$ the bucket's label (given by the first b bits of x);
- set $R[j] := max(R[j], \rho(x))$, where $\rho(x)$ is the leftmost 1-bit position of x truncated of its first b bits;

Compute $Z := (\sum_{j=1}^{m} 2^{-R[j]})^{-1}$, it is the harmonic mean of $2^{R[j]}$;
return $E := \alpha_m m^2 Z$
with $\alpha_m := (m \int_0^\infty (log_2(\frac{2+u}{1+u}))^m du)^{-1}$.

III. Description of the Sliding HyperLogLog algorithm

To adapt the HyperLogLog algorithm to the data stream context, some additional time information must be maintained. In fact, the objective is to estimate at any time t the number of distinct flows seen over the last w unit of time, for any duration w smaller than the time window W. A possible solution is to perform the HyperLogLog algorithm on the packets received in the concerned duration. But this naive solution implies an exhaustive storage of the packets received over the last time window because we have no a priori information about w and t. So this is clearly an unscalable solution. The algorithm we propose aims to maintain a short list of packets. A packet consists of a pair $< t_i, R_i >$, where the t_i is the arrival time of the packet and R_i is the position of the leftmost 1-bit in the binary representation of the hashed value associated to this packet. The idea is to store only packets that are useful to the computation of the crucial parameter R, which is the biggest value of R_i. A packet is stored only if it is a possible maximum over a future window of time. The list of stored packets is called List of Future Possible Maxima (LFPM). It is updated in the following way:

For each received packet $< t_k, R_k >$ do

- Delete old packets (packets with $t_i < t_k - W$) from the list LPFM
- Delete packets with $R_i \leq R_k$ from the LPFM
- Add $< t_k, R_k >$ to the LPFM

This simple update mechanism is analogous to the one used in [3].
An example of a LPFM is given in Figure 2

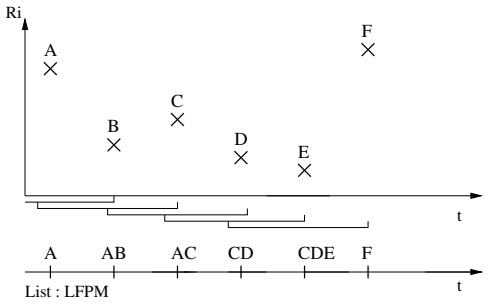

Fig. 2. Example of a List of Future Possible Maxima (LFPM)

The stochastic averaging process consisting of splitting the packets into m buckets, in HyperLogLog algorithm, remains unchanged. It means that we maintain one LPFM per bucket. Each list is updated independently. To estimate, at a time t, the number of flows received over the last w units of time, we first extract the LPFM packets with a timestamp$\geq t - w$. Then we compute the highest R_i among those packets. These operations are performed separately for each bucket. The harmonic mean and the estimate of the number of flows are the same as in HyperLogLog algorithm.

A. The accuracy of the algorithm

The update mechanism of the LPFM has no impact on the computation of R. It is simply an efficient method that allows us to compute exactly R at any time t and over any duration w. It has the advantage to store only a short list of packets. As the rest of the algorithm is unchanged, it is

clear that adding the sliding window has no impact on the accuracy of the HyperLogLog algorithm. It means that the Sliding HyperLogLog algorithm gives the same estimate as the HyperLogLog algorithm applied on the packets received in $[t-w,t]$.
One can conclude that the standard error on the estimate given by the Sliding HyperLogLog algorithm is of about $1.04/\sqrt{m}$. Notice that the Sliding LogLog algorithm outperforms the Sliding MinCount algorithm [3], which has a higher standard error of about $1.3/\sqrt{m}$.

B. The memory usage

In the HyperLogLog algorithm, the total used memory is given by the m registers $R[i], for i \in [1,m]$. Thus the required memory equals $mln_2ln_2(n/m)$ bits, where n is the real number of flows. As explained before, to design an algorithm adapted to data stream, we need to store more details about the traffic. The required memory in the Sliding HyperLogLog algorithm is given by the size of the m lists of possible future maxima (LPFM), that we denote L_n^{tot}. More precisely, a LPFM has L_n packets, where a packet is a pair $< t_i, R_i >$. As R_i depends on the number of flows n, the size of the list also depends on this parameter.
In practice we need a 4 byte timestamp, R_i can be stored on 1 byte for a data stream of several million flows. Therefore the size of the total used memory in bytes equals $5\ L_n^{tot}$. L_n^{tot} can be given by the following equation $L_n^{tot} = m\ E(L_n)$, where $E(L_n)$ is the mean size of the LPFM in packets. To conclude, the Sliding HyperLogLog algorithm requires a memory of $5\ m\ E(L_n)$ bytes.
To estimate $E(L_n)$, we compare the proposed algorithm to the Sliding MinCount algorithm [3]. This latter algorithm is based on an order statistics observable which is the smallest hashed value. The main difference between the two algorithms is the update mechanism of the LPFM. In the Sliding MinCount algorithm, a packet is remains in the LPFM if its associated hashed value H_i is smaller than the hashed value of the current received packet H_k. However in the Sliding HyperLogLog algorithm, the condition concerns R_i and R_k which represent the position of the leftmost 1-bit. Notice that these two conditions are not independent. In fact $R_i > R_k$ implies $H_i < H_k$. However, we can have $H_i < H_k$ and $R_i = R_k$. For instance one can take $H_i = 0010010$ and $H_k = 0010110$. Thus the packet $< t_k, H_k >$ is maintained in the list for the Sliding MinCount algorithm and is deleted from the LPFM in the Sliding HyperLogLog algorithm. So the mean size of the LPFM, $E(L_n)$ is clearly smaller in the Sliding HyperLogLog algorithm. In [3] Fusy et al show, using a classical result in combinatorics, that when n gets large, $E(L_n) \to ln(n/m)$. Such result is difficult to establish for the Sliding HyperLogLog algorithm, as the same tools are not adapted in our context. However, as the LPFM is smaller, we can at least state that for the Sliding HyperLogLog algorithm, $ln(n/m)$ becomes an upper bound to $E(L_n)$, when n gets large.

IV. Simulations and experimental results

In this section, the Sliding HyperLogLog algorithm is validated using two kinds of data. First, we simulate a generic data stream. It can be seen as a synthetic traffic, where elements represent packets. Second, a real traffic trace issued from campus networks is considered. For both cases, the objective is to count the number of flows over a sliding time window. Recall that the Sliding HyperLogLog algorithm allows us to estimate at any time the number of distinct flows that have been seen over the last w unit of time, for any w smaller than the time window W.

A. The accuracy of the algorithm

We focus here on the performance of the algorithm in terms of the accuracy of the estimation of the number of flows.

Synthetic traffic

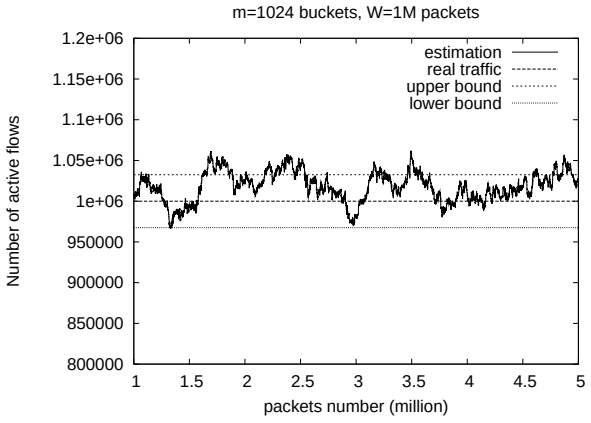

Fig. 3. Estimation of the number of active flows in a logical window, on a synthetic traffic

First, we consider a simple case, where a flow consists of only one packet. So the traffic is only composed of packets belonging to distinct flows. We simulate a data stream of 5 millions packets. The time window W is taken equal to 1 million packets. Every 100 packets, we answer the following query: "How many flows have been seen among the last 1 million received packets?". The estimated number of flows is plotted in Figure 3. It is compared to the real value (1 million in this case). One can notice that the error on the estimation of the number of flows is most often within the theoretical bound of 3.25% ($= 1.04/\sqrt{m}$, m being equal to 1024). It sometimes slightly exceeds this value. So the experiments are in agreement with the intuition that adding the sliding window mechanism does not affect the accuracy of the original HyperLogLog algorithm.

Real traffic

The same experiments are performed on a traffic trace issued from campus networks: “Abilene” trace. This traffic trace has been captured in June 2004. It was found at the URL:
http ://pma.nlanr.net/Traces/Traces/long/ipls/3
Some characteristics of this trace are given in the following table:

TABLE I
CHARACTERISTICS OF ABILENE TRAFFIC TRACE

Nb. IP packets	Nb. TCP Flows	Duration
42 939 465	1 389 805	8 minutes

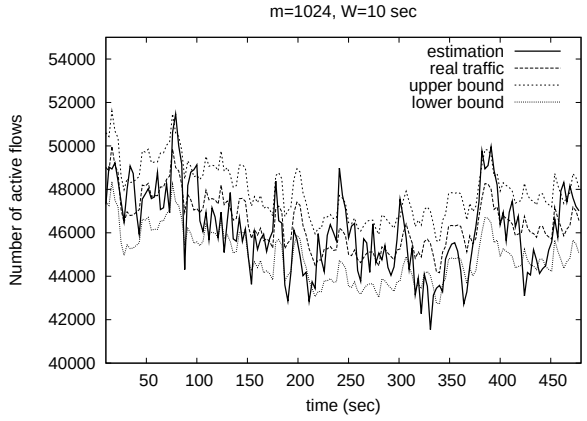

Fig. 4. Estimation of the number of active flows in a sliding time window

In Figure 4, a physical time window W is considered. It is taken equal to 10 seconds. Every second, we estimate the number of distinct flows received over the last 10 seconds. The number of buckets, m, equals 1024. Experimental results show that the error on the estimation of the number of flows is close to its expected value. Notice that unlike the simulated traffic, the real number of flows over the last 10 seconds is variable, but it does not vary too much.

A logical time window is used in Figure 5. Time is here computed as the number of received packets. We consider the same query as for the synthetic traffic: “How many flows have been seen among the last 1 million received packets?”. This query is asked every $100,000$ packets. One can simply check that experimental results are in agreement with theory.

B. The cost of the algorithm

The main advantage of the Sliding HyperLogLog algorithm consists of the capacity to cope with a data stream of any duration. Moreover, it can answer a query with a variable duration w (bounded by the time window W). To provide these options, it needs additional ressources compared to the original version: the HyperLogLog algorithm. The major difference

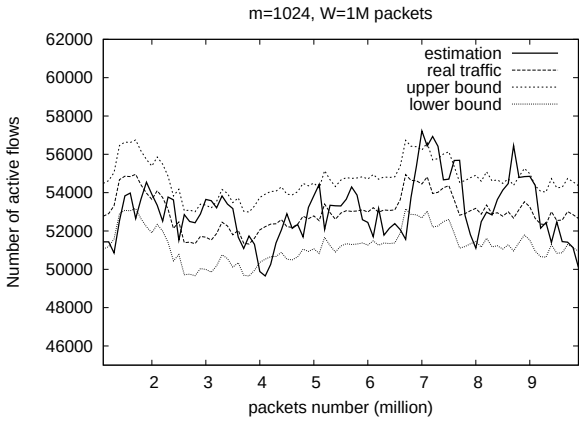

Fig. 5. Estimation of the number of active flows in a logical window

between these two algorithms is the presence of the List of Future Possible Maxima (LFPM) in Sliding HyperLogLog algorithm. The total used memory is given by the size of the m LFPM, referred to as L_n^{tot} in Section III. Recall that we have one LFPM per bucket. The execution time of the algorithm also depends on the size of the LFPM, as all elements of this list are updated for each received packet.

Figures 6 and 7 show the evolution of L_n^{tot} for respectively synthetic and Abilene traffic. According to experiments, L_n^{tot} has a small variance.

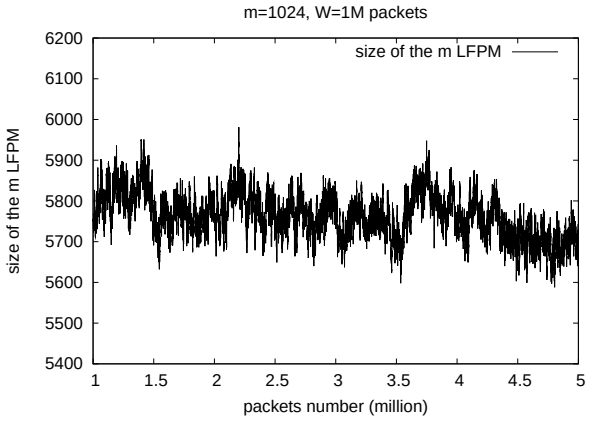

Fig. 6. Size of the list LFPM, Synthetic traffic

In figures 8 and 9, the distribution of the LFPM, L_n, is plotted. One can notice that L_n takes small values for both synthetic and Abilene traffic, with a respective average of 5 and 3. This difference can be explained by the fact that the number of flows in a time window is not the same in both cases. For the synthetic traffic it is given by the number of

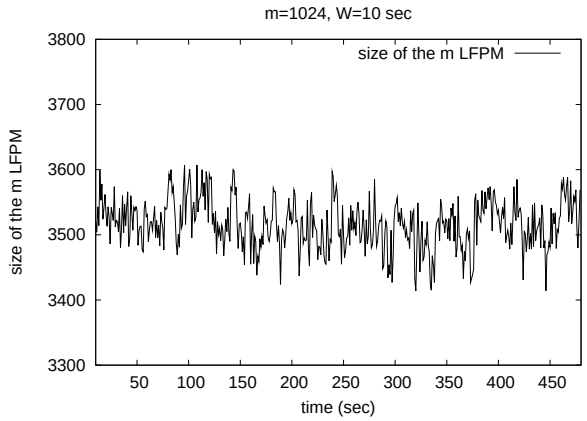

Fig. 7. Size of the list LFPM, Abilene traffic

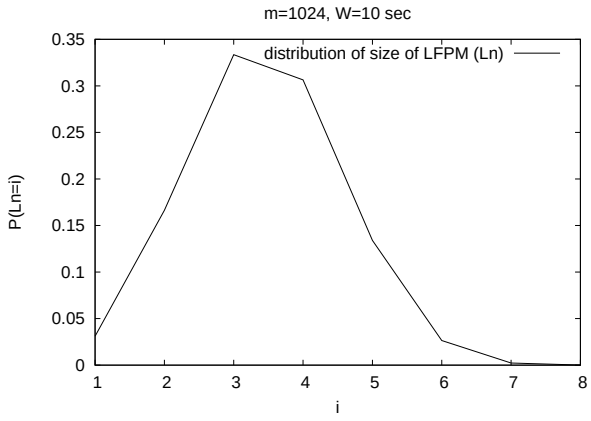

Fig. 9. The distribution of the size of the list LFPM, Abilene traffic

packets in a time window : 1 million flows, and for abilene traffic, there is about $45,000$ flows in a time window of 10 seconds.

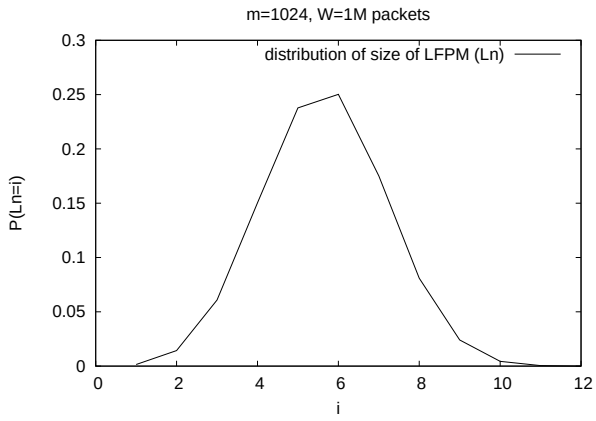

Fig. 8. The distribution of the size of the list LFPM, Synthetic traffic

In Figures 10 and 11, the mean size of the LFPM, $E(L_n)$, is plotted for different values of m, for respectively synthetic and Abilene traffic. $E(L_n)$ is compared to the theoretical asymptotic (when the total size of flows n is very large) mean size of the LFPM in the Sliding MinCount algorithm presented in [**?**]. For synthetic traffic n equals 1 million and for Abilene traffic, n is given by the real number of flows in a time window of 10 seconds. It is supposed here to be constant (equal to $45,000$) as it has a very small variance. One can notice that $E(L_n)$ is almost usually smaller than the asymptotic mean equal to $ln(n/m)$. So experimental results confirm the fact that Sliding HyperLogLog needs a smaller memory than Sliding MinCount algorithm. Moreover, as the LFPM is updated in the same way for both algorithms, a smaller LFPM implies a shorter updating time.

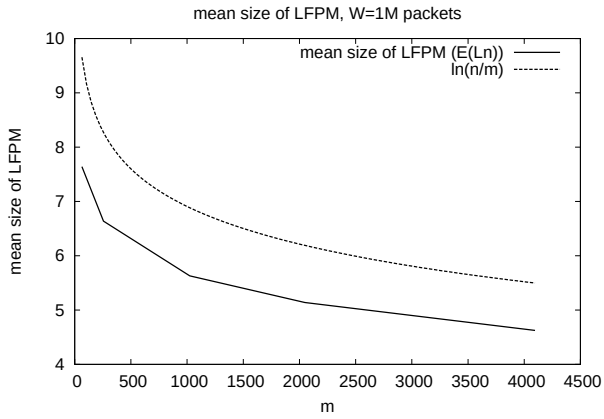

Fig. 10. Impact of m on the mean size of the list LFPM, using a synthetic traffic

As mentionned in Section III, L_n represents the number of elements in the LFPM. An element consists of a pair $< t_i, R_i >$, R_i is encoded in $ln_2ln_2(n/m)$ bits in Sliding HyperLogLog algorithm. However for Sliding MinCount algorithm, R_i corresponds to the hashed value which needs $ln_2(n/m)$ bits. We can conclude that the total memory size is smaller in Sliding HyperLogLog algorithm. This latter has also a better accuracy than Sliding MinCount algorithm, with a relative error of $1.04/\sqrt{m}$ instead of $1.3/\sqrt{m}$.

The total execution time of the two algorithms can not be compared because the updating time of the LFPM is certainly shorter for Sliding HyperLogLog algorithm, but this algorithm has an additionnal processing step which consists of deducing R_i from the hashed value for every received packet. In the

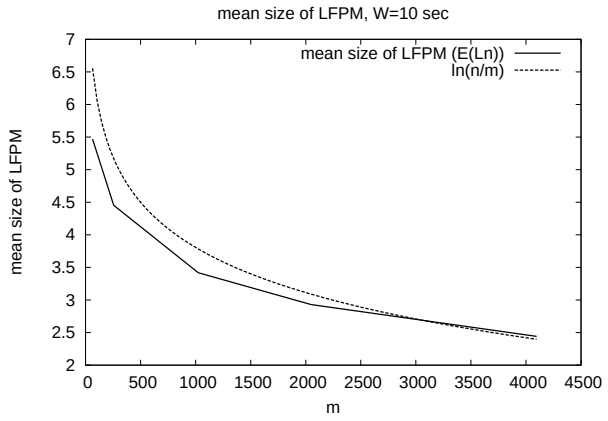

Fig. 11. Impact of m on the mean size of the list LFPM

Sliding MinCount algorithm, the hashed value is directly used to update the LFPM. So we have clearly a tradeoff between the execution time and the memory usage.

REFERENCES

[1] C. Estan and G. Varghese and M. Fisk, *Bitmap algorithms for counting active flows on high speed links*. Internet measurement conference, 2003.

[2] P. Flajolet and G. N. Martin, *Probabilistic counting algorithms for data base applications*. Journal of Computer and System Sciences 31, 1985.

[3] E. Fusy and F. Giroire, *Estmating the number of active flows in a data stream over a sliding window*. Proceedings of the Ninth Workshop on Algorithm Engineering and Experiments and the Fourth Workshop on Analytic Algorithmics and Combinatorics (ANALC0), January 2007.

[4] M. Durand and P. Flajolet, *LogLog Counting of Large Cardinalities*. Proc. European Symposium on Algorithms 2003.

[5] P. Flajolet and E. Fusy and O. Gandouet O. and F. Meunier, *Hyper-LogLog: the analysis of a near-optimal cardinality estimation algorithm*. Proceedings of the 13th conference on analysis of algorithm (AofA 07), 2007.

[6] F. Giroire, *Order Statistics and Estimating Cardinalities of Massive Data Sets*. In Proceedings of Discrete Mathematics and Theoretical Computer Science, 2005.