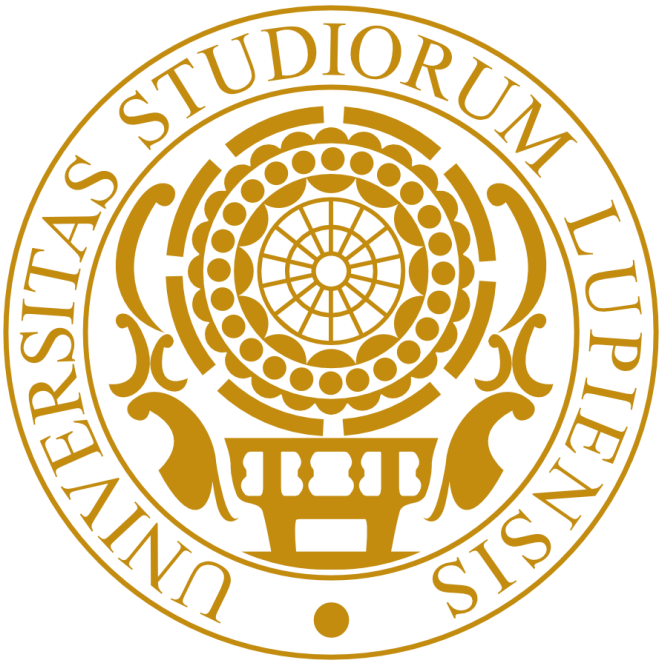


HeavyTracker

Heavy hitter detection Algorithm

Laila Haddad

25/1/2025

Introduction

Heavy-hitter detection, which aims to find the set of flows that constitute most of the high-speed network traffic, is a fundamental task in network measurement. The flow can be defined as a set of packets identified by the same flow label, e.g., source IP/Port, destination IP/Port. The flow frequency [1] refers to the number of packets in a flow. For detecting heavy hitters, our focus will be on the elephant flows whose frequencies exceed a predefined threshold t. There are multiple algorithms that can assist with this task (Count-min sketch, CM-CU, HeavyKeeper, e.g), but each one introduces new problems. For example, both Count-min and CM-CU are suited for memory-constrained environments [2] with their compact sketch structure, but they suffer an overestimation problem due to hash collisions. HeavyKeeper solves that problem by introducing exponential decay, which selects elephant flows by allowing the mouse flows to decay and make room for elephant flows. However, the decay probability decreases exponentially, which means a medium flow might take the place of an elephant flow only because it was captured earlier, and it is not likely to be overridden. Additionally, if a heavy hitter gets repeatedly replaced, its calculated frequency will be lower than its true frequency, causing an underestimation problem. HeavyTracker has the best of both worlds; it combines the compact data structure, exponential decay, and fingerprint tracking with higher accuracy.

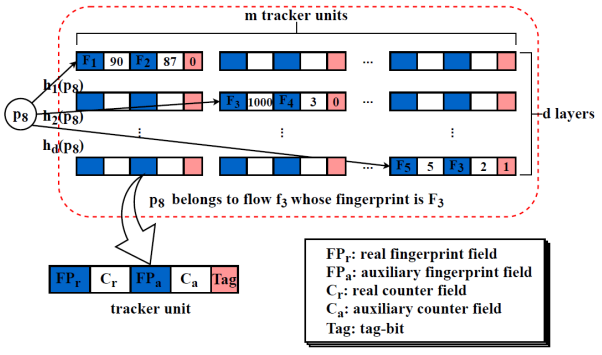
Algorithm Analysis

The purpose of the algorithm is to track the elephant flows in each tracker unit with higher accuracy and fewer omissions. It is challenging because the elephant flows are noised by many mouse or medium-sized flows hashed in the same tracker unit.

The algorithm processes each incoming data packet sequentially, maintaining an approximate frequency count for heavy hitters using a sketch structure. Each row has a hash function that maps the incoming flows to a tracker unit. The hash functions ensure a uniform distribution of keys, reducing hash collision.

Each tracker unit contains:

- FPr: real fingerprint field
- Cr: real counter field
- FPa: auxiliary fingerprint field



- Ca: auxiliary counter field
- Tag: tag bit

A fingerprint is a unique identifier of the flow. The real (FPr, Cr) pair is responsible for storing the first detected heavy hitter fingerprint and count. The auxiliary pair (FPa, Ca) stores an additional heavy hitter, in case the real pair was already occupied. The tag bit stores the current update mode.

Exponential expulsion

For each flow currently preserved in the tracker unit, it has an exponential probability of being expelled by a new incoming item that is different from it. The expulsion probability is exponentially smaller as the current recorded frequency grows larger, similarly to HeavyKeeper.

Update modes

The algorithm has 2 update modes, A and B, distinguished by the tag bit.

- **Mode A**: the initial tracker unit mode, where no heavy hitter was recorded. The unit accepts 2 potential heavy hitters in both counter pairs, and the counters are incremented according to the incoming packets.
    - If the incoming flow has a different fingerprint than the existing fingerprints, we let the auxiliary flow be expelled with probability $P_{plus} = \frac{q}{b^{Ca}+c} + \gamma$.
    - If the condition **Ca > Cr** is achieved, the pairs are swapped. The real pair must always contain the flow with the higher frequency.
    - Finally, when Cr reaches the threshold t, the tag bit is set to 1, indicating that a heavy hitter was found, and the update mode must change to B.
    - FPr value is copied to FPa that will act as a static record, where the fingerprint will be preserved as a heavy hitter. The counters are reset to 0, and FPr to empty, to accept a potential new heavy hitter.
    - Both counters are reset because we do not care for keeping track of the heavy hitter count, as it has been added to the heavy hitter list already.
- **Mode B:** indicated by the tag bit. Only FPr, Cr and Ca are tracked. Since FPa is occupied with the identified heavy hitter, Ca is free to be used as a decay indicator for FPr.
    - If FPr in empty, the incoming flow is moved to it.

- If the incoming flow's fingerprint = FPr, both counters are incremented.
- If the fingerprints are different, exponential expulsion is applied to Cr, giving the chance for a protection heavy hitter to take its place.
- We do exponential decay on Ca with probability $P_{decay} = \frac{1}{b^{Ca}}$
- If Ca decays down to 0, we replace FPr with the new flow's fingerprint and set Ca=1.

The flow is reported as heavy In **Mode A**, if Cr hits t. In **Mode B**, if an incoming flow matches the static FPa or it matches FPr where Cr ≥ t.

Parameter Optimization and Genetic Algorithm (GA)

The parameters q, b, c, γ mentioned in the exponential expulsion are experimental parameters that can be tuned using a genetic algorithm to improve the accuracy of flow expulsion. The algorithm used is Stochastic-Universal Selection, which is a selection technique resembling a roulette wheel to mix and match random combinations of the parameters, and test their fitness.

- The algorithm runs HeavyTracker with the parameter combination repeatedly over a test dataset, in order to minimize the error between the real and estimated frequencies as such:

$$\text{Fitness function} = -\frac{1}{m}\sum \left| \widehat{n}_{max} - n_{max} \right|$$

- When the fitness function converges, the output parameters are then used to run HeavyTracker on the full dataset, with optimal identification.

The parameter ranges

- B: [1, 1.1] and
- C: [0..10] to make the value of the denominator is always larger than 1.
- γ: (0..0.001] so the probability of being expelled doesn't evaluate to 0.
- q: 0, 1 − γ so Pplus ∈ (0, 1].

Implementation and Testing

The algorithm and parameter optimization were implemented using c++. Count-min sketch was also implemented to compare the results and accuracy of the algorithms.

Heavy tracker

Each tracker unit evaluates to 17 bytes total. The number of layers in the sketch is D = 3 , M the number of units in each layer, and the threshold t were chosen depending on the size of the dataset.

```
struct TrackerUnit {
    std::string realFP;
    int realC = 0;
    std::string auxiliaryFP;
    int auxiliaryC = 0;
    bool tag = false;
};
```

The dataset was taken from ScienceDirect , a dataset that captures seven days of monitoring data from eight servers hosting more than 800 sites across a large campus network. The dataset was then split into 3 sets, a set that contained a million rows to compare HeavyTracker and Count-min, and 2 smaller sets of 150k rows for training and testing the efficiency of GA.

Genetice algorithm

After several trials, these were the chosen hyper parameters

Population size = 20 , Max generations = 30

Crossover rate = 0.7 , Mutation rate = 0.03

Count-min

The implemented code was merged from several online sources.[3] [4]

Performance analysis

With the hyperparameters mentioned above, it took the algorithm approximately 70 minutes to complete the generations, using the 150k row test data. D = 1 to simplify the error calculation process.

M	D	Best Fitness	b	c	q	gamma
10000	1	-459939	1.02196	8.35294	0.32548	3.23412e-05

The same parameters are then used with the HeavyTracker to compare it to count-min using D = 3, and a million rows of data.

Test 1: Threshold = 1,000,000, True number of heavy hitters: 149

M	HeavyTracker	Count-min
10000	**151**	**158**
1000	**219**	**227**
100	**1792**	**2314**

There is a clear overestimation problem for both algorithms, the smaller M gets, which can be explained by collisions.

Test 2: Threshold = 10,000, True number of heavy hitters: 1474

M	HeavyTracker	Count-min
10000	**1487**	**1508**
1000	**1797**	**2360**

In both tests, when the suitable M is chosen, HeavyTracker's accuracy is ≃98.9%, while count-min is ≃96.02%

Scalability

The parameters generated from the 150k row dataset held up well for both the 150k rows and the million rows dataset and gave near accurate results. HeavyTracker used up 500KB to process that entire dataset accurately, while count-min used 120KB only.

Conclusion

HeavyTracker provides a highly accurate and scalable solution for heavy-hitter detection in high-speed networks. By combining the strengths of compact data structures, exponential decay, and fingerprint-based tracking, it effectively addresses the overestimation and underestimation issues found in other algorithms like Count-min Sketch and HeavyKeeper. Here’s a well-structured conclusion for your paper:

The experiments demonstrate that HeavyTracker achieves remarkable accuracy, even under memory-constrained conditions, with an accuracy of approximately 98.9% when an appropriate M is chosen. In contrast, Count-min Sketch, while slightly less accurate at 96.02%, offers better memory efficiency.

Parameter optimization using a Genetic Algorithm proved effective in fine-tuning the algorithm's performance. The parameters generated using a smaller dataset were successfully scaled to larger datasets, proving scalability and adaptability of HeavyTracker. Despite the added computational cost of parameter tuning, the improved accuracy makes it a strong candidate for real-world applications.

references

[1] J. Yu *et al.*, “HeavyTracker: An Efficient Algorithm for Heavy-Hitter Detection in High-Speed Networks,” in *2022 IEEE 28th International Conference on Parallel and Distributed Systems (ICPADS)*, Nanjing, China: IEEE, Jan. 2023, pp. 362–370. doi: 10.1109/ICPADS56603.2022.00054.

[2] G. Cormode and S. Muthukrishnan, “An Improved Data Stream Summary: The Count-Min Sketch and its Applications”.

[3] “https://github.com/barrust/count-min-sketch ”.

[4] “https://github.com/alabid/countminsketch ”.