Heavy Hitter Stability
6.829 Final Project

Sitan Chen, Qinxuan Pan

1. PROBLEM

The heavy-hitter detection problem asks to find a list of frequently appearing elements in a stream of data. Network managers might want to gather frequent IP addresses on packets going through a router because they want to 1) detect dramatic changes in traffic pattern (e.g. popularities of services, search words on Google [4], etc.), 2) detect services that are potentially under DDOS attack, or 3) (especially in data centers) perform traffic enginnering [1].

Consider an existing router without the hardware capabilities (e.g. parallelized hashing operations) to run heavy-hitter algorithms efficiently. Suppose the operator wants to enable heavy-hitter detection, but upgrading the hardware is costly. One solution is to implement the algorithm in software, which provides a lot of memory and programming flexibility. However, the software might not be able to process information at the same rate at which the hardware processes it.

2. APPROACH

We deal with problems like the above by proposing a general method to implement heavy-hitter computation while straddling across two components of the same architecture, one with small memory but fast processing (e.g. hardware), and vice versa for the other (e.g. software). Our main idea is that the list of heavy hitters tends to be locally stable over time. Consequently, we can maintain a small ***core list*** of IPs (and their counts) in hardware, obtained as the estimated list of heavy hitters at some point in recent past, and use it to filter out most of the incoming packets. The filtered packets will be processed only by the hardware, which increments the corresponding counts. The software only needs to deal with the remaining packets, alleviating the discrepancy in speed. The core list will only be refreshed upon significant decline in its ability to filter out packets.

We start by defining some key quantities.

2.1 Cover Size

The concept of cover size helps us gauge how big a core list we need to filter out a desired percentage of packets. We start with a naive definition.

DEFINITION 1. *Given a stream of m packets (i.e. m pairs of source/destination IPs) over a time interval. Associate a node to each IP, and set its weight to be the number of times the IP appears in the stream. We define the* ***(node-based) P% cover size*** *to be the minimum number of nodes necessary to account for P% of the total node weights. We define the* ***cover density*** *to be the ratio between the cover size and the total number of unique IPs encountered in the stream.*

While this definition seems to capture the notion of heavy-hitters, it suffers from the problem that the resulting list tends to be too big, and includes too many junk IPs (IPs that appear only very few times). For example, if the source of each of the m packets is the same IP, while the destinations are all unique, then intuitively, the common source IP should be the only heavy-hitter. However, to cover, say, 95% of the total node weights, it will be necessary to include most of the unique destination IPs, as the common source IP only takes up 50% of the total weight.

This leads to the alternative definition that we will focus on.

DEFINITION 2. *Given the same setup as above, we set the weight of each edge to be the number of times the IPs corresponding to its two end nodes appear as the source/destination pair (or vice versa) of some packet. We define the* ***(edge-based) P% cover size*** *to be the minimum number of nodes necessary to cover P% of the total edge weights, where an edge is covered as long as either one of its end points is covered. The* ***cover density*** *is defined similarly as above.*

If we set the core list in the hardware to be such a cover list of IPs (for a time interval in recent past), it turns out that we will be able to filter out most of the incoming packets if we filter a packet whenever *either its source or destination is on the core list.* It might appear that some counts are ignored. For example, consider a packet whose source is on the core list, but not its destination. Then, the packet will be filtered. While the

count for the source IP is incremented at the hardware, the statistics for the destination IP is updated by neither hardware nor software.

It turns out that those missing counts don't matter too much in the quest of detecting heavy-hitters. The point is that we have in mind the following "traffic graph" to approximate reality: there is a small set of core IPs (e.g. popular services), and a large set of junk IPs (e.g. individual users), and most of the traffic will be between a core IP and a junk IP. In this case, we obviously just want to pick out the core IPs as heavy-hitters. If the source IP of a packet is in the core, then most likely the destination IP is a junk IP, and can therefore be safely ignored. Our experimental results in the next section suggest that this approximation is reasonable.

2.2 Stability

The concepts quantifying stability help us make decision regarding updating the core list in hardware. Again, there are two plausible definitions.

DEFINITION 3. *Consider two time intervals T_1 and T_2. Let L_1 and L_2 be the (edge-based) P% core list for packets in T_1 and T_2, respectively (we mostly consider $P = 95$). We define the* ***core correlation*** *between intervals T_1 and T_2 to be the percentage of IPs in L_1 that remain in L_2.*

The core correlation essentially measures how much the list of heavy-hitters changes over time. It gives us an idea of how many IPs to replace when we update the core list. However, we use another quantity to determine when to update. Recall that, ultimately, the purpose of the core list is to filter out as many future packets as possible. It does not have to track the current heavy-hitters perfectly, as it is not going to be used as the answer to a heavy-hitter query directly (See the concluding section on how to answer queries).

DEFINITION 4. *Given the same setup as above, we define the* ***cover correlation*** *between T_1 and T_2 to be the percentage of packets in T_2 that can be filtered out based on L_1.*

Suppose we last updated the core list to be L_1 at the end of T_1, then we will update it again only upon significant cover correlation decay. It turns out that cover correlation is reasonably long-lasting.

3. EVALUATION

The primary source of data used in this work was the CAIDA 2016 Anonymized Internet Traces Dataset. Specifically, we conducted experiments on a roughly one-hour interval (13:00 - 14:03) of passive trace data taken from an Equinix data center in Chicago on January 21, 2016. The primary piece of information we used from this dataset was the stream $\mathcal{S}$ of source-destination IP pairs that occured in this time interval.

3.1 Cover Size

Recall that the notion of edge-based cover size defined above amounts to a minimal partial vertex cover. We first discuss some implementation details regarding approximating this quantity (its exact computation is NP-Complete). Upon constructing the adjacency list associated to $\mathcal{S}$, our algorithm initializes a priority queue with a key associated to each node v of the graph and corresponding priority equal to v's degree in the graph, and maintains a "mass counter" corresponding to the total edge weights covered by our core list so far. At each step, the algorithm pops the node u with highest priority, removes all appearances of this node within the adjacency list, updates the priorities of its neighbors v in the priority queue by decrementing their degrees by the original weight of edge uv in the graph, and increments the mass counter by the priority of the popped node u. This loop terminates as soon as the mass counter equals at least 95% of the sum of all edge weights, and the core list is all popped nodes. Put more succinctly, our algorithm is the greedy algorithm for computing an approximately minimal partial vertex cover.[1]

To study cover size, we isolated a single minute of data, 13:00-13:01. The traffic for this minute consisted of roughly 30 million packets and 750000 distinct IP addresses among both source and destination IPs. We observed that the 95% cover size is roughly 8500 (out of 750000), giving a cover density of 1.14%. We likewise analyzed time scales of 10 and 30 seconds and observed average cover densities of 3.55% and 2.97% in these regimes. It is unsurprising that the 95% cover density shrinks as time scale increases because we expect the concentration of heavy hitters to manifest more clearly over longer time.

To compare, we also computed the naive node-based 95% cover size, simply by aggregating a list of all distinct IPs and greedily picking the most frequently occurring ones until only 5% of all IPs, counted with multiplicity, remain. In the three regimes above, average node-based cover density was 23.06%, 19.14%, and 14.69%, much higher than the edge-based case.

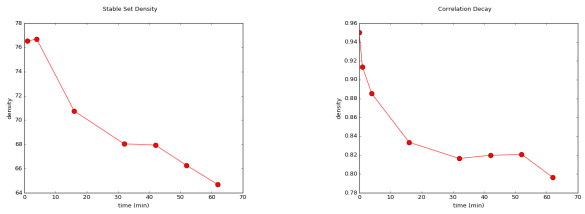
3.2 Stability

To study stability, we analyzed several one-minute windows from 13:00 to 14:03 in the dataset, specifically 13:00-13:01, 13:01-13:02, 13:04-13:05, 13:16-13:17, 13:32-13:33, 13:42-13:43, 13:52-13:53, and 14:02-14:03.

[1]While it is known classically that this greedy algorithm obtains only an $O(\log n)$ approximation for minimum vertex cover, its output on the sorts of graphs we are interested in as discussed in the previous section is closer to our notion of heavy hitters than, say, the output of the randomized 1/2-approximation algorithm for minimum vertex cover.

We computed the 95% (edge-based) cover list for each window, and the sizes of the intersections of the list for the 13:00-13:01 window with every later list. The results are shown in Figure 1a. The core correlations between the first window and the later windows exhibit a decline from 76% to roughly 64%. The immediate drop to 76% suggests that the optimal cover list changed by roughly a quarter in a minute. Does it imply that we need to update the core list in hardware every < 1 minute?

For that, we studied the cover correlations in the same regime. We computed the 95% cover list L_1 for the first window and, for each subsequent window, determined the percentage of packets covered by L_1 (i.e. percentage of total weights of all edges in that window incidental to some node in L_1). The results are plotted in Figure 1b. Remarkably, cover correlation remains above 82% for almost the entire hour, only exhibiting a dip to slightly below 80% for the final window. It shows that, for example, L_1 can still filter out roughly 4 in every 5 packets, even an hour later.



(a) Core correlation (b) Cover correlation

4. CONCLUSION

Overall our results suggest that our intuitions for cover size and stability are accurate within time scales on the order of minutes or hours, and useful for formulating potential implementations for heavy hitter detection. That the 95% cover size over a minute of traffic comprises fewer than 10000 distinct IPs implies that the router could support such a core list, together with counter for each IP on it, even with fewer than 100 kilobytes of memory. Judging from our results on cover correlation decay, the packet-filtering rate would be above 85% even if we update the core list every 10 minutes (see Figure 1b). As sketched in Section 1, the unfiltered packets would then be processed in software, either using a naive histogram or, if sublinear query time is desired, sketching methods like count-min sketch augmented by the dyadic trick [2].

There are several options on how to update the core list. For instance, the top 10000 IPs from among both hardware and software counts could be selected to comprise the new core list. Alternatively, based on our core correlation decay results (see Figure 1a), we know that the cores of minute-long windows spaced ten minutes apart have roughly 72% intersection, suggesting we could also simply demote the bottom 28% from the old core list and promote the top 28% based on the software-maintained counts.

To answer a query for heavy-hitters, we simply return the IPs with top counts, from both hardware (the core list) and software.

Ultimately, the choices of parameters and update methods should be tuned based on available hardware memory, speed ratio between hardware and software, query frequency, and router traffic pattern. In any case, our framework should be viewed not as an alternative to, but as a way of augmenting any existing heavy hitter algorithm. Our filtering approach is just a way to lighten the load placed on whatever streaming algorithm by essentially removing a significant fraction of the stream from consideration, exploiting the heavy-hitter stability. It opens the door to implementing more sophisticated sketching techniques (e.g. [3]) in software that would have been impossible in purely hardware-based approaches. It is a general approach that can be useful upon new generations of hardware/software, and across various traffic characteristics (it doesn't matter if the traffic is bursty, we still ensure that software only needs to deal with a small fraction of all packets that are actually processed, instead of dropped, by the hardware).

Experimentation code and additional plots can be found at `https://github.com/sitanc/heavyhitters/`.

4.1 Contributions

The two authors contributed roughly equally to coming up with the approach, the quantities to focus on, and the analysis of experimental data. The first author was responsible for the code, and the second author was responsible for the presentation slides. We are also thankful for helpful discussions with the course instructors.

5. REFERENCES

[1] M. Al-Fares, S. Radhakrishnan, B. Raghavan, W. College, N. Huang, and A. Vahdat. Hedera: Dynamic flow scheduling for data center networks. In Proceedings of NSDI 2010, San Jose, CA, USA.

[2] Graham Cormode and S. Muthukrishnan. An improved data stream summary: the count-min sketch and its applications. *J. Algorithms*, 55(1):58–75, 2005.

[3] Kasper Green Larsen, Jelani Nelson, Huy L. Nguyen, and Mikkel Thorup. Heavy hitters via cluster-preserving clustering. In *Proceedings of the 57th Annual IEEE Symposium on Foundations of Computer Science* (FOCS), 2016.

[4] Rob Pike, Sean Dorward, Robert Griesemer, and Sean Quinlan. Interpreting the data: Parallel analysis with Sawzall. Scientific Programming, 13(4):277–298, 2005.