Sapienza University of Rome

# Big Data Computing Third Homework

Andrea Fioraldi 1692419

January 19, 2020

## Exercise 1

FOR each node $x$ DO
  IF $x$ in $C$ THEN
    $Mcur(x)$ = concatenation of $k$ bitmasks each
      with 1 bit set ($P$(bit $i$) = $.5^{i+1}$)
FOR each distance $h$ starting with 1 DO
  $stop = true$
  FOR each node $x$ DO $Mlast(x) = Mcur(x)$
  FOR each edge $(x, y)$ DO
    $Mcur(x)$ = ($Mcur(x)$ BITWISE-OR $Mlast(y)$)
    IF $Mlast(x) \neq Mcur(x)$ DO
      $stop = false$
  IF $stop = true$ DO
    return $h - 1$

The above algorithm is the ANF-0 algorithm proposed in [1] with the addition of a termination condition. For the notation refer to [1].
The algorithm stops when all the node has a stable collection of $k$ bitmasks, so when an iteration of the algorithm doesn't change any $Mcur(x)$ for all node $x$ in the graph.
This happens, thanks to the approximation of the reached nodes using the bitmasks, when all the nodes reached all the other nodes in the graph. The algorithm evolves hop by hop, so when the last node $x$ stabilize its bitmasks we have that the distance between $x$ and the last node added to $Mcur(x)$ is the approximate maximum distance between two nodes in the graph.
The diameter of a graph [2] is defined as the maximum distance (shortest path) between two nodes in a graph and so the distance $h - 1$, when the algorithm stabilizes all the

bitmasks of all nodes, is an approximated solution to the problem.
Note that, is we use a bitvector in which each bit represents if a node is in the set or not instead of $k$ bitmasks (and so $O(n^2)$ space), we have the exact diameter.

## Exercise 2

I propose a variant of Reservoir sampling [3].

```
1  class StreamSample(object):
2      def __init__(self, s):
3          self.s = s
4          self.hashes = [create_rand_hash() for i in range(s)]
5          self.sample = [(None, MAX_INT)] * s
6
7      def update(self, i):
8          for j in range(self.s):
9              h = self.hashes[j](i)
10             if h <= self.sample[j][1]:
11                 self.sample[j] = (i, h)
12
13     def get(self):
14         return list(map(lambda x: x[0], self.sample))
```

A property of MinHash is that a shingle $y$ is mapped to the minimum position of the randomly permuted bitvector with a probability 1 over the cardinality of the document. We can view the elements processed by the algorithm like shingles in MinHash and so define that a hash $h_j := \pi_j(x)$ created with `create_rand_hash` is the position of an element $x$ in the random permutation $\pi_j$. Note that we can also define $h_j$ as an ideal hash function, the algorithm will work in the same way.
Thanks to the above property, the probability that an item $i$ is in a specified sample bucket $j$ (so that its position is the min in $\pi_j$ considering all the previous $n$ distrinct elements) is $P\big[i = sample_j\big] = \frac{1}{n}$.
We consider only *distinct* items because duplicates are hashed to the same value and don't have the chance to be inserted. In detail, from the second time that the item $i$ is processed, the hash of the item in the bucket is lower or equal than $h_j(i)$ and so it has no chance to be the minimum.
Assume that we are in the case in which $n \geq s$.
The outcome of the hashes are independent events, so the probability that an item $i$ is not in all of the $s$ buckets is $\left(1 - \frac{1}{n}\right)^s$.
From this follows that:

$$P\big[i \in sample\big] = 1 - P\big[i \notin sample\big] = 1 - \left(1 - \frac{1}{n}\right)^s \approx 1 - \left(1 - \frac{s}{n}\right) = \frac{s}{n}$$

The approximation can be easily derived from the Taylor expansion [4] of $(1+x)^a$ that works when $|x| < 1$ and $|ax| << 1$. In our case, $x = -\frac{1}{n}$, the first is trivially $n > 1$, the second is $\left|s * \left(-\frac{1}{n}\right)\right| << 1 \rightarrow \frac{s}{n} << 1 \rightarrow n >> s$.

More $n$ is greater than $s$, better is the approximation because the truncated terms in the series become progressively smaller.

### A note about the sample

Of course, an item $i$ can be in more than one bucket. This is a possible scenario but has a negligible probability because we are assuming that random permutations are different and ideal and so the probability that $i$ have the min position for two different $\pi_j$ is small and decrease with $n >> s$.
Despite this, the algorithm requirements do not specify that the sample should be a set so I assume that the above scenario is tolerable.

## Exercise 3

$$\mathbb{E}\left[\hat{x}^T\hat{y}\right] = \mathbb{E}\left[\frac{1}{m}x^T S^T S y\right] = \frac{1}{m}\mathbb{E}\left[x^T\right]\mathbb{E}\left[S^T S\right]\mathbb{E}\left[y\right]$$

The expectation of $S^T S$ is just a matrix containing the expected values of the elements of the matrix $S^T S$.
Each element is the product of the $i$-th row of $S^T$ and the $j$-th column of $S$.
So the value of each element of the expectation is:

$$\mathbb{E}\left[S^T_{i,*}S_{*,j}\right] = \mathbb{E}\left[\sum_{k=1}^{m} S^T_{i,k}S_{k,j}\right] = \begin{cases} \sum_{k=1}^{m}\mathbb{E}\left[S^T_{i,k}S_{k,j}\right] = 0 & i \neq j \\ \mathbb{E}\left[\sum_{k=1}^{m} S_{k,i}S_{k,j}\right] = \mathbb{E}\left[\sum_{k=1}^{m} 1\right] = m & i = j \end{cases}$$

Consider $a, b \in \{-1, 1\}$, two possible values in the matrix $S$.
The case when $i \neq j$ is derived from the product $a*b$ can be 1 or -1 with equal probability (when the signs are discordant, so with probability 2/4, the result is -1 otherwise 1). The case $i = j$ is due to the fact that when $a = -1$ we have that $a * a = 1$.
So the matrix $M$ can be defined as:

$$M_{d\times d} = \mathbb{E}\left[S^T S\right] \text{ such that } M_{i,j} = \begin{cases} 0 & i \neq j \\ m & i = j \end{cases}$$

Trivially, follows that:

$$\mathbb{E}\left[\hat{x}^T\hat{y}\right] = \mathbb{E}\left[x^T\right]\frac{1}{m}M_{d\times d}\,\mathbb{E}\left[y\right] = x^T I_{d\times d} y = x^T y$$

## Exercise 4

I designed an algorithm that uses $O(log_2(n) * log_2(log_2(n)))$ memory.

```
class StreamSum(object):
    def __init__(self, n):
        self.n = n
        self.ln = int(math.log(n, 2))
        self.R = [0]*self.ln

    def update(self, i):
        r = 0
        for b in range(self.ln):
            if not (i & (1<<b)): r += 1
            else: break
        for b in range(self.ln):
            if i & (1<<b):
                self.R[b] = max(self.R[b], r)

    def get(self):
        s = 0
        for b in range(self.ln):
            s += 2**self.R[b] * 2**b
        return s
```

Basically, for each bit from position 1 to $log_2(n)$, it counts how many distinct numbers in the stream have this bit set to 1. The approximate sum of the distinct numbers in the stream is the sum of the powers of 2 correspondings to each bit multiplied for the numbers of distinct numbers that have the bit set to 1. The approximate counting of distinct numbers is done using Flajolet–Martin [5].

### Why this works?

Consider a stream without duplicates, our problem now can be solved with a simple sum of all the incoming numbers. I'll show an algorithm that is equivalent to the sum and so that solves our problem without duplicates, then I'll generalize in the case of a stream with duplicates.
For each bit position b, I count all the numbers that have the bit at position b set using an array of number R. Each entry of R is incremented when a number has the bit corresponding to the entry set to 1. An example run is:

- 6: 0110, $R_0 = 0, R_1 = 1, R_2 = 1, R_3 = 0$;
- 9: 1001, $R_0 = 1, R_1 = 1, R_2 = 1, R_3 = 1$;
- 10: 1010, $R_0 = 1, R_1 = 2, R_2 = 1, R_3 = 2$;

Doing this, we increase the counters corresponding to the powers of 2 that compose a number (eg. 6=4+2). The sum of all of these powers of 2 is equivalent to the sum of all the numbers due to the distributive property of the sum. In fact, in the final round of the example, we have that $2^0 * 1 + 2^1 * 2 + 2^2 * 1 + 2^3 * 2 = 1 + 4 + 4 + 16 = 25$ that is equal to $6 + 9 + 10 = 25$.
In order to have this working in the case of a stream with duplicates, I rely on Flajolet–Martin to count the approximate number of distinct numbers that have a bit set to 1.

This simple generalization is at least as valid as the algorithm of Flajolet–Martin is, so don't expect a high accuracy but this is the best that I can do with such small memory requirement.

## References


[1] C. R. Palmer, P. B. Gibbons, and C. Faloutsos, "ANF: a fast and scalable tool for data mining in massive graphs," in *Proceedings of the eighth ACM SIGKDD international conference on Knowledge discovery and data mining*, (New York, NY, USA), pp. 81–90, ACM Press, 2002.

[2] "Distance (graph theory) - Wikipedia." `https://en.wikipedia.org/wiki/Distance_(graph_theory)`. Accessed: 2020-1-17.

[3] "Reservoir sampling - Wikipedia." `https://en.wikipedia.org/wiki/Reservoir_sampling`. Accessed: 2020-1-15.

[4] "Binomial approximation - Wikipedia." `https://en.wikipedia.org/wiki/Binomial_approximation#Using_Taylor_Series`. Accessed: 2020-1-13.

[5] "Flajolet–Martin algorithm - Wikipedia." `https://en.wikipedia.org/wiki/Flajolet%E2%80%93Martin_algorithm`. Accessed: 2020-1-11.

# Appendix

Appendix code for Exercise 1 (not MinHash or an ideal hash, just if you want to run the code):

```
MAX_INT = 257
def create_rand_hash():
    a = random.randint(0, MAX_INT-1)
    b = random.randint(0, MAX_INT-1)
    return lambda x: (x + a) % b
```

Code to test the Exercise 4:

```
def rand_stream(size, n):
    for i in range(size):
        yield random.randint(0, n-1)

N = 1024
ss = StreamSum(N)
rs = set()
for i in rand_stream(1000, N):
    ss.update(i)
    rs.add(i)

print("Estimated sum :", ss.get())
print("Exact sum     :", sum(rs))
```

An example run:

```
Estimated sum : 349525
Exact sum     : 334596
```