Online and streaming algorithms

Motivation

- In *a lot* of different applications, input is submitted in a *streaming* fashion, and decisions should be taken *online* (i.e. before seeing next items)
  - financial data (stock quotes, orders, sales, …)
  - data coming from sensors, monitors, satellites, cameras, …
  - internet traffic, routing, …
  - supporting database updates
  - …
- Even if data is available *offline* (e.g. on disk), it can be so big that it can only be “scanned through”

Two problematics

- Online algorithms: How much does the *online* mode of processing (as opposed to *offline*) affect the solution of a problem?

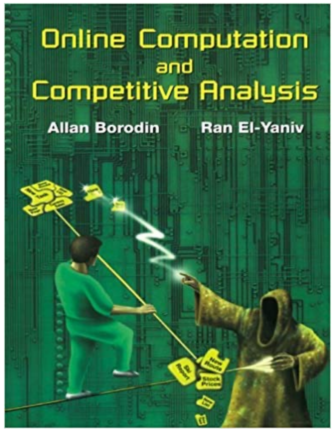
- Streaming algorithms: Can we efficiently analyze big data streams online using small amount of working memory?
  - *Example*: collecting statistics about connections to Google website

Offline vs online

- ***Offline***: all input data is available beforehand

- ***Online***: data is input by units in a stream
  - a unit should be processed “immediately” (otherwise it cannot be accessed later)
  - no information available about items to come
  - at any moment, the algorithm has “done all the work” for the data that has been received up to now
  - *real time*: $O(1)$ *worst case* time spent on each unit

Competitive analysis



- σ : sequence of *requests*
- An online algorithm *alg* is *c-competitive* iff its cost (e.g. time) *alg*(σ)≤*c*·*offline*(σ), for all sequences σ, where *offline*(σ) is the cost of an optimal offline algorithm
- Remarks:
  - σ is the only input
  - we are interested in the worst-case (over all inputs)
  - concept of *adversary*

Ski rental problem

- renting skis costs \$10/day, buying skis costs \$100
- you don't know how many days you are going to ski. Rent or buy?
  - *Technical point*: you can buy before you know if you will ski the next day (e.g. end of previous day), but you can rent the same day (e.g. on your way to ski lift in the morning)

Ski rental problem

- renting skis costs $10/day, buying skis costs $100
- you don’t know how many days you are going to ski. Rent or buy?

- *Offline algorithm*: if you ski $k < 10$ days, then rent (spend $\$10 \cdot k$), otherwise buy (spend $\$100$)

Ski rental problem

- renting skis costs \$10/day, buying skis costs \$100
- you don't know how many days you are going to ski. Rent or buy?

- *Online algorithm*:

Ski rental problem

- renting skis costs \$10/day, buying skis costs \$100
- you don't know how many days you are going to ski.
  Rent or buy?

- *Online algorithm*:

  - *Example*: ski for 3 days, then buy ⇒ you can spend $3 \cdot 10 + 100 = \$130$ but if you stop skiing after 4 days, you could have spent only \$40. $c \geq \frac{130}{40} = 3.2$

Ski rental problem

- renting skis costs \$10/day, buying skis costs \$100
- you don’t know how many days you are going to ski. Rent or buy?

- *Online algorithm*: rent for 10 days then buy
- this algorithm is 2-competitive

Ski rental problem

- renting skis costs \$10/day, buying skis costs \$100
- you don't know how many days you are going to ski. Rent or buy?

- *Online algorithm*: rent for 10 days then buy
- this algorithm is 2-competitive

- Can we do better?

Ski rental problem

- renting skis costs $10/day, buying skis costs $100
- you don’t know how many days you are going to ski. Rent or buy?

- *Online algorithm*: rent for 10 days then buy
- this algorithm is 2-competitive

- Can we do better?
- **NO** with a deterministic algorithm
  - buy after $k < 10$ days $\Rightarrow c = \frac{10k+100}{10k} = 1 + \frac{10}{k} > 2$
  - buy after $k > 10$ days $\Rightarrow c = \frac{10k+100}{100} = 1 + \frac{k}{10} > 2$

Ski rental problem

- renting skis costs \$10/day, buying skis costs \$100
- you don't know how many days you are going to ski. Rent or buy?

- *Online algorithm*: rent for 10 days then buy
- this algorithm is 2-competitive

- Can we do better?
- **NO** with a deterministic algorithm
- **YES** with a probabilistic algorithm: there exists an *expected* 1.58-competitive online algorithm

Probabilistic ski rental: intuition

- If I buy skis after day j ($j \leq 10$), the worst scenario is that I ski only j days
- However, for different j's worst scenarios are different
- If we choose j randomly, we avoid the impact of a single worst scenario

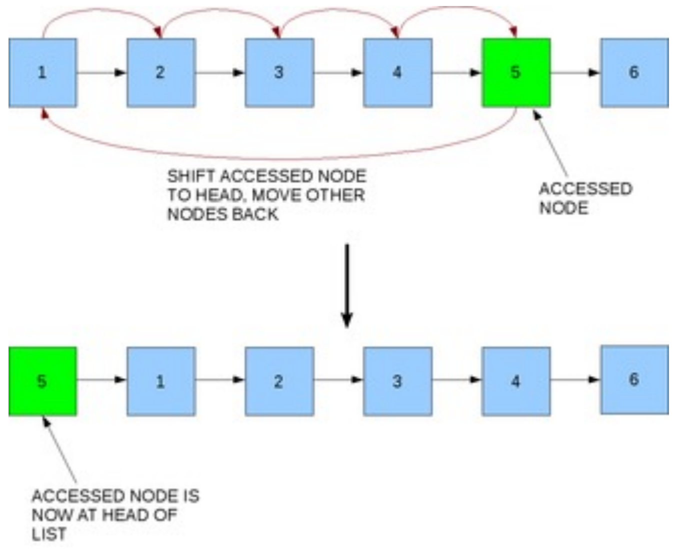
- *Example*: buy skis after 8 days with proba $1/2$ and after 10 days with proba $1/2$. Then expected competitiveness c is
  - if I ski <8 days, then $c = 1$
  - if I ski 8 days, then $c = \frac{1}{2} \cdot \frac{80+100}{80} + \frac{1}{2} \cdot 1 = 1.625$
  - if I ski 9 days, then $c = \frac{1}{2} \cdot \frac{80+100}{90} + \frac{1}{2} \cdot 1 = 1.5$
  - if I ski ≥10 days, then $c = \frac{1}{2} \cdot \frac{80+100}{100} + \frac{1}{2} \cdot \frac{100+100}{100} = 1.9$

Probabilistic ski rental: general case

- Let
  - rental costs 1 per day
  - buying skis costs B
  - $T \geq 1$ be the # of days of skiing in the input
  - the algorithm buys skis after j days ($0 \leq j \leq B$), with proba p_j
- If $T \leq B$, then $c_{\leq B} = \sum_{i \leq T} p_i \cdot \frac{i+B}{T} + \sum_{T < i \leq B} p_i \cdot 1$
- If $T > B$, then $c_{>B} = \sum_{i \leq B} p_i \cdot \frac{i+B}{B}$
- $c_{opt} = \min\{\max_T\{c_{\leq B}, c_{>B}\} | \sum p_i = 1\} \approx \frac{e}{e-1} \approx 1.58$

Self-organizing lists [Sleator&Tarjan 1985]

- a set of elements is stored in a singly linked list
- a sequence of `access` operations given online
- accessing i-th element *e* in the list costs i
- after accessing e, we are allowed to move it (towards the head) *with no cost*
- *goal*: minimize the total cost of all accesses


1
2
3
4
5
6
SHIFT ACCESSED NODE TO HEAD, MOVE OTHER NODES BACK
ACCESSED NODE
5
1
2
3
4
6
ACCESSED NODE IS NOW AT HEAD OF LIST

Self-organizing lists: offline solution

- offline algorithm DF (*Decreasing Frequency*): sort elements in the non-increasing order of access frequency

- DF is optimal if no moves are allowed
- Is it also optimal if moves are allowed? In general not. (E.g. accessing last element several times)
- Computing the minimum cost is complicated (NP-complete)

Self-organizing lists: online algorithms

- MF (*Move-to-Front*): after accessing an element, move it to the head of the list
- T (*Transpose*): after accessing an element, exchange it with the immediately preceding item
- FC (*Frequency count*): Maintain a count of each element, initially 0, incremented by 1 when the element is accessed. Maintain the list in the non-increasing order of counts.

Self-organizing lists: results

- T does not have a constant competitive ratio
  - *Counter-example*: alternatively access the two last elements of the list

- FC does not have a constant competitive ratio
  - *Counter-example*: access the 1st element $k > n$ times, the second $(k-1)$ times, … and the last $(k-n+1)$ time. The list is then never modified. The cost is

$k + 2(k-1) + 3(k-2) + \cdots + n(k-n+1) \geq (k-n)(1+2+3+\cdots+n) = \Theta(kn^2)$.

  With MF the cost is $k + [2 + (k-2)] + [3 + (k-3)] + \cdots = kn$
  - *Corollary*: MF can be a factor of $O(n)$ better than FC

Self-organizing lists: Move-to-Front

- *Theorem* [Sleator&Tarjan 85]: MF is 2-competitive

Self-organizing lists: Move-to-Front

- *Theorem* [Sleator&Tarjan 85]: MF is 2-competitive

- *Remarks*:
  - This is best possible among deterministic algorithms
  - Also, the best possible among probabilistic algorithms if each next request can be chosen depending on the previous moves (adaptive online adversary)
  - However, this can be improved to ~ 1.6-competitive if the whole sequence of requests is fixed in advance (oblivious adversary), but there is a lower bound of ~ 1.5-competitive
  - The picture is different for the average-case complexity, when we access randomly according to access probabilities $(p_1, p_2, \dots, p_n)$. The measure is then the *expected* cost. In this case,
    - DF *is* an optimal algorithm
    - MF is no better than T [Rivest 1976]

Разборчивая невеста (Secretary problem)

- A princess chooses a husband
- n candidates are presented one-by-one (n is known to the princess)
- for any two candidates, the princess is able to say who is better
- the order of candidates is random
- each candidate is either definitely accepted or definitely rejected
- *goal*: maximize the probability to choose the best

Разборчивая невеста (Secretary problem)

- A princess chooses a husband
- n candidates are presented one-by-one (n is known to the princess)
- for any two candidates, the princess is able to say who is better
- the order of candidates is random
- each candidate is either definitely accepted or definitely rejected
- *goal*: maximize the probability to choose *the best*

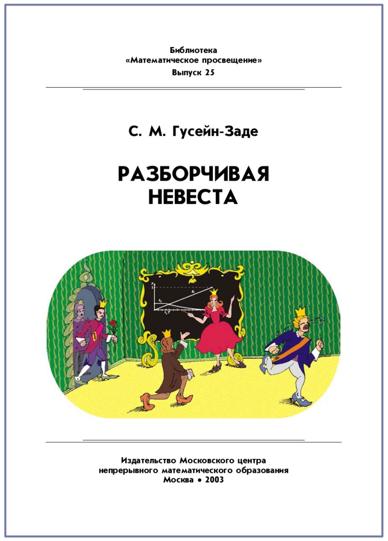
…
Чтоб в одиночестве не кончить веку,
Красавица, пока совсем не отцвела,
За первого, кто к ней присватался, пошла:
И рада, рада уж была,
Что вышла за калеку.

И.А.Крылов, Разборчивая невеста

Разборчивая невеста (Secretary problem)

- *NB*: not searching for best competitiveness

- *Optimal strategy*: reject $(r-1)$ candidates, then select the first one which is better than any of them
- Analysis shows that $r = 1/e \cdot n \approx 0.368 \cdot n$ (rule of 37%)
- The probability of selecting the best candidate is $1/e$ as well



Библиотека
«Математическое просвещение»
Выпуск 25

С. М. Гусейн-Заде

РАЗБОРЧИВАЯ
НЕВЕСТА

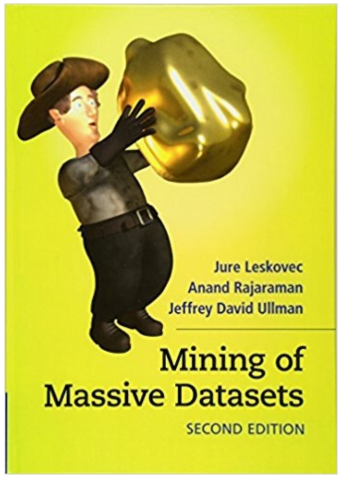
Издательство Московского центра
непрерывного математического образования
Москва • 2003

Mining big data streams

Examples of big stream sources

- Sensor data
- Image data
- Internet and web traffic
- Phone calls
- Web searches
- streams of customers/purchased items/…
- …

- Common characteristics:
  - data comes at a (very) high rate
  - data cannot be stored
  - data should be processed online, in low time and per item
  - approximate answers are often ok



Problem 1: Sampling a stream

- *General idea*: sample a stream (e.g. consider 1/10 of the items) in hope that the sampled stream will have similar properties
- Cannot be done by simply sampling 1 over 10 items!
- Why?

Problem 1: Sampling a stream

- *General idea*: sample a stream (e.g. consider 1/10 of the items) in hope that the sampled stream will have similar properties
- Cannot be done by simply sampling 1 over 10 items!
- Why? Assume we have a stream where s items occur once and d occur twice, and we want to estimate the fraction of repeated elements (right answer $\frac{d}{s+d}$).
- If we sample each element with $p = \frac{1}{10}$, then out of d repeated items, $\frac{d}{100}$ will occur twice in the sample, $\frac{81 \cdot d}{100}$ will disappear, and $\frac{18d}{100}$ will become unique
- The estimate will be $\frac{\frac{d}{100}}{\frac{s}{10}+\frac{18d}{100}+\frac{d}{100}} = \frac{d}{10s+19d}$

Sampling a stream (cont)

- How to solve this problem?

Sampling a stream (cont)

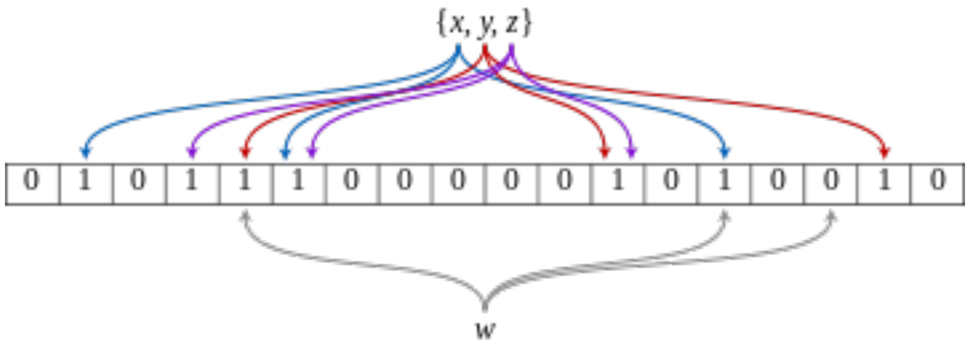
- How to solve this problem? Use hash functions!
- Hash items to numbers, sample those whose hash ends with 0 (in decimal notation)

Problem 2: Checking if an element has been “seen before”

Problem 2: Checking if an element has been "seen before"

- Can be done with Bloom filters!
  - reminder: hash-based bitmap data structure
  - takes 1-2 bytes per *distinct* element, admits false positives

  - INSERT(k): set $h_i(k) = 1$ for all i
  - LOOKUP(k): check $h_i(k) = 1$ for all i



Problem 3: Keeping multiplicities of elements and maintaining “heavy hitters”

- Find frequent (often occurring) elements in a stream

- *Example*: frequently viewed products in an online shop

Problem 3: Keeping multiplicities of elements and maintaining "heavy hitters"

- *Counting Bloom filters*: same as Bloom filters but replace individual bits by counters
- when inserting an element, increment corresponding counters by 1

- INSERT(k): set $h_i(k) = h_i(k) + 1$ for all i
- DELETE(k): set $h_i(k) = h_i(k) - 1$ for all i
- LOOKUP(k): check $h_i(k) > 0$ for all i

Count-Min sketch

- What if we want to estimate the multiplicities (number of occurrences) of elements of a multi-set stored in a counting Bloom filter?
- *Example:* $m = 8,\ C[0..7]$

	a	b	c	d	e	f
h_1	0	2	3	7	4	5
h_2	4	5	1	2	1	7

$b\ a\ d\ a\ e\ f\ \uparrow\ c\ a\ \uparrow\ \ldots$

$\#a?\ \#e?\ \#f?\quad \#c?\ \#e?\ \#f?$

Count-Min: operations

- UPDATE(k): $C[h_i(k)] \leftarrow C[h_i(k)] + 1$ for all i
- $\hat{f}(k) = \min_i h_i(k)$

Count-Min sketch: analysis

- *Theorem*: if $d = \log_2 \frac{1}{\delta}$ and $m = \frac{ed}{\varepsilon}$, then

$$P[\hat{f}(k) \geq f(k) + \varepsilon n] \leq \delta,$$

where d is the number of hash functions, $f(k)$ is the true count of k and n is the total number of elements in the stream

Proof: $C[h_i(k)] = f(k) + X_i(k)$

$$E[X_i(k)] = \frac{1}{m}\sum_{l \neq k} f(l) + \sum_{j \neq i} \frac{1}{m} \sum_{k} f(k) \leq \frac{d}{m} n = \frac{\varepsilon}{e} n$$

$$P[X_i(k) > \varepsilon n] < \frac{1}{e}$$

Markov inequality
$$P[X \geq a] \leq \frac{E[X]}{a}$$

Since $\hat{f}(k) = f(k) + \min_i X_i(k)$, we have

$$P\left[\hat{f}(k) - f(k) \geq \varepsilon n\right] \leq e^{-d} = \delta$$

Count-Min: properties

- Total space $m = \frac{e}{\epsilon} \log \frac{1}{\delta}$
- Bound in Theorem is in terms of n
- CM-sketch also applies to increments > 1
- Decrements ("deletions") are also supported provided that counters remain non-negatives
- Count-Min [Cormode&Muthukrishnan 2005] sketch vs Spectral Bloom filters [Cohen&Matias 2003]
- For positives increments there exists a better strategy: *conservative update*

Heavy hitters

- *Heavy hitters*: elements occurring at least n/k times (k parameter)
- Assume we want to output all elements that occur $n/50$ of times. Set $\varepsilon = 1/100$, i.e. $m = 271 \cdot \log \frac{1}{\delta}$. Then we will output all desired elements, but also some elements occurring less, but not less than $n/100$, with $p = 1 - \delta$
- Cf. also Misra-Gries algorithm, Boyer-Moore majority vote algorithm

What if n is not known in advance?

- *Idea*: Maintain a min-heap of current frequent items, update after each element
- After processing each element x, estimate $\hat{f}(x)$
- If $\hat{f}(x) \geq \varepsilon n$ (n current stream size), insert x to the heap with value $\hat{f}(x)$ (or update if it was already there)
- If the smallest value of the heap (computed in $O(1)$) is $< \varepsilon n$, delete it from the heap
- At the end, output all elements of the heap

Problem 4: Counting distinct elements in a stream

- *Count-distinct problem*: count the number of distinct elements in a (very large) stream
- *Examples*:
  - estimating the cardinality for memory allocation (Bloom filter)
  - unique users of a web site,
  - distinct IP addresses (routers, web servers, …)
  - detecting DoS attacks
  - number of distinct words (k-mers) in a (streamed) text (DNA sequence)
  - …
- How can we beat the naïve solution?
  - **count approximately!**
  - **use probabilities!**

Approximate counting (Morris 1977)

- Robert Morris (Bell Labs): maintain the logs of a very large number of events in small registers
- Algorithm:
  - maintain K that stores (approx value of) $\log(n)$, i.e. size of K is $\log\log(n)$
  - initialize $K = 0$
  - when a new event arrives, increment K **with probability $\mathbf{2^{-K}}$**
- K reaches a value k after $1 + 2 + 4 + \cdots + 2^{k-1} = 2^k - 1$ steps, i.e. $E[2^K - 1] = N$ (N is true count)
- $\sigma^2 = N(N-1)/2$ i.e. $\sigma \approx N/\sqrt{2}$ (70% of N)

Try this

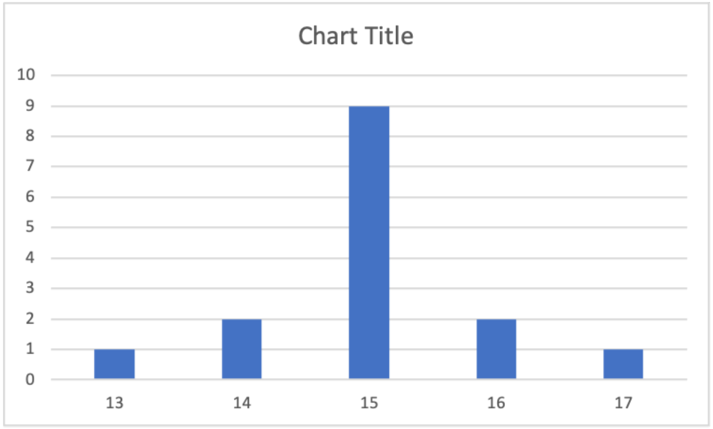
- Implement this and run several times, acquire statistics:

```
pick $N$ (e.g. $N = 15$)
$c = 0$
for $i = 1$ to $2^N$ do
        increment $c$ with probability $2^{-c}$
print $c$ // compare $c$ with $N$
```

Try this

- Implement this and run several times, acquire statistics:

```
pick $N$ (e.g. $N = 15$)
$c = 0$
for $i = 1$ to $2^c$ do
        increment $c$ with probability $2^{-c}$
print $c$ // compare $c$ with $N$
```



How to improve?

- *Improvement 1*: change base of logarithms from 2 to some smaller b ($b > 1$), count $\log_b N$
  - increment K with probability b^{-K}
  - $E[b^K - 1] = (b - 1)N$, $\sigma^2 = (b - 1)N(N - 1)/2$
  - price: space increased from $\log\log n$ to $\log\log_b n$

- *Improvement 2*: keep m counters instead of just one, then compute the average/median …

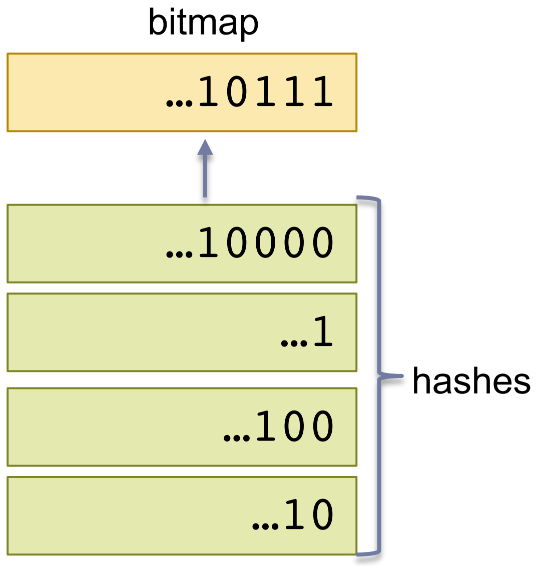
Counting distinct elements in a stream [Flajolet & Martin 85]

Main idea:

- hash elements into (binary) numbers $[0..2^L - 1]$ using a good hash function h
- hashes …xxx1 are expected to occur ½ time, …xxx10 are expected ¼ time, …xxx100 1/8 time, etc.
- max$_k$[hash …10^i are *all* observed for $0 \leq i \leq k$]+1 is a good indication of $\log N$ (N nb of distinct elements)

Flajolet & Martin algorithm: implementation

- maintain a bitmap of size L, set all bits to 0
- for each hash, compute position of the rightmost 1 and set the corresponding bit of bitmap to 1
- let i be the position (from right) of the rightmost 0 in bitmap
- then the nb N of unique elements is estimated as $2^{i-1}/\varphi$, φ=0.77351..



Intuition: if $i \ll \log N + 1$, then it is almost certainly 1; if $i \gg \log N + 1$, then *i*-th bit of bitmap is almost certainly 0

- *Pb*: big variance of results (accuracy within a factor of ~ 2)
- *Solution 1*: run the algo m times, then take average/median/combination (accuracy $O(1/\sqrt{m})$)
- *Solution 2* [FM]: *stochastic averaging*

  use first k bits of hash to dispatch elements into $m = 2k$ bins, then average;

  accuracy $\approx 0.78/\sqrt{m}$