Algorithms, Randomness and Big Data

Francesco Silvestri

Department of Information Engineering

University of Padova

silvestri@dei.unipd.it

Who am I? Francesco Silvestri

Experience:

- PhD University of Padova in Computer Engineering
- Visiting scholar University of Texas at Austin
- Post-doc University of Padova and IT University of Copenhagen
- Now: associate professor at University of Padova

Research:

- Big data algorithms: how to efficiently extract information from big-data?
- High performance algorithms: how to exploit modern computer architecture for big-data?

Outline of the talk

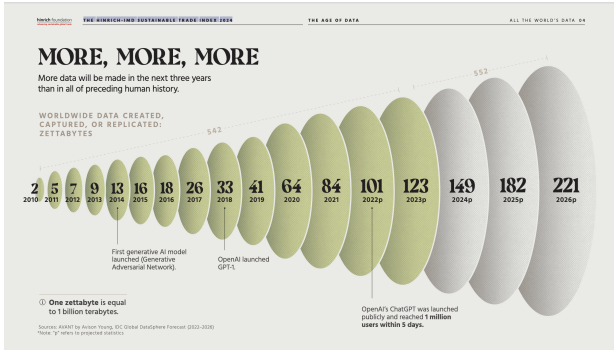
- Introduction to the Big Data phenomenon
- Data stream: counting and randomness
- Research in algorithm design

Big data

Big Data Phenomenon

Source: Data Never Sleeps Project

Big Data Phenomenon



From: *Hinrich Foundation - The Age of Data 2024*

How big is 221 ZB?:

- 1 ZettaByte (ZB) $=$ 1 trillion GB $= 10^{12}$ GB.
- Downloading 221 ZB at 1Gb/s takes $>$ 54 million years.
- Streaming UltraHD resolution for entire EU population for 7 years.

Big Data Phenomenon

The world continuously collects huge amounts of:

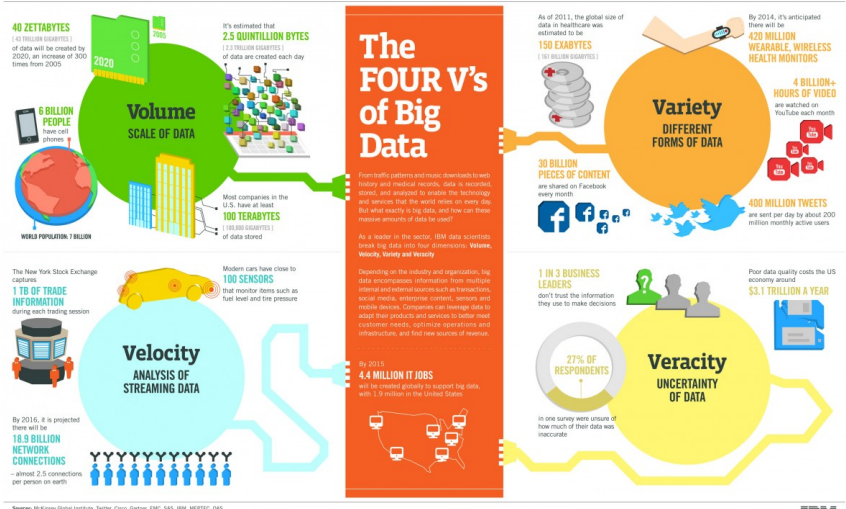
- Physical data: from sensors, telescopes, particle physics experiments.
- Biological/medical data: from genetic studies, patient monitoring, epidemic evolution analyses.
- Human activity data: from social networks, mobile devices, internet/web traffic, IoT systems.
- Business data: from online stores, customer profiling, bank/credit-card/financial services, quality-of-service monitoring.

Big Data Phenomenon

Why is DATA has been growing so much?

- Technological progress:
  - Growth of storage capacity
  - Growth of comunication bandwith
  - Growth of computing capacity
- Reduction of ICT costs
- Pervasiveness of digital technologies: scientific research, health, business, politics, social interactions, ...

Computing Challenges



Source: IBM Big Data & Analytics Hub

Computational Challenges

- **Volume:** processing huge datasets poses several challenges and requires a data-centric perspective.
- **Velocity:** sometimes, the data arrive at such a high rate that they cannot be stored and processed offline. Hence stream processing is needed.
- **Veracity:** large datasets coming from real-world applications are likely to contain *noisy, uncertain data*, hence accuracy of solutions must be reconsidered.
- **Variety:** large datasets arise in *very different scenarios*. More effective processing is achieved by adapting to the actual characteristics of data.

Computational Challenges

The above issues require a

paradigm shift w.r.t. traditional computing.

To tackle the above challenges effectively, one needs:

- Modern computational frameworks, for instance
  - Parallel programming frameworks for dealing with volume
  - Data streaming frameworks for dealing with velocity
- Advanced algorithmic ideas:
  - Give up with exact and deterministic solutions.
  - Data are noisy: why spending time on exact solutions?
  - Leverage on approximation and randomized techniques.

Data stream and randomness

Dealing with volume and velocity

Typical scenario:

- The data to be processed arrive as a continuous stream
  - because they are generated by some evolving (possibly endless) process;
  - or because their massive volume discourages random accesses.
- Therefore: data analysis must happen on the fly using limited memory, without storing all data for subsequent offline processing.

Some applications

Network management:

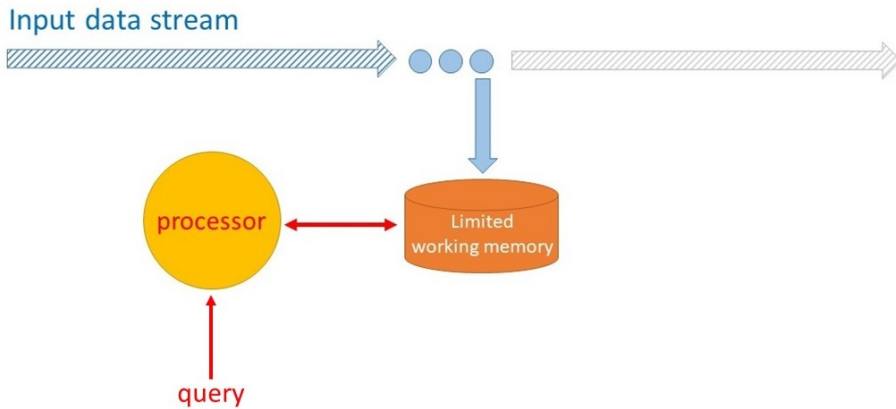
- Stream: packets routed through a router.
- Task: gather traffic statistics (e.g., average number of connections/second to same IP address)

Internet of Things:

- Stream: data generated by thousands/millions of sensors.
- Tasks: learn from data, outlier detection, gather statistics,...

Streaming Model

- Sequential machine with limited amount of working memory.
- Input provided as a continuous (one-way) stream.



Streaming Model

Let $\Sigma = x_1, x_2, \ldots x_n \ldots$ denote the input stream received sequentially, where x_i is the i-th element received. Upon receiving x_n:

- Suitable data structures stored in the working memory are updated (**UPDATE task**)
- If required, a solution for the problem at hand, relative to the input set $x_1, x_2, \ldots, x_n$ is computed from the data stored in the working memory (**QUERY task**)

The following **Key Performance Indicators** are usually considered

- KPI 1: Size s of the working memory (aim: $s \ll |\Sigma|$)
- KPI 2: Number p of sequential passes over Σ (aim: $p = 1$)
- KPI 3: Update time T_u per item (aim: $T_u = O(1)$)
- KPI 4: Query time T_q to return a solution after seeing $x_1, \ldots, x_n$ (aim: T_q independent of n)

Streaming Model

Algorithm Design Techniques

- Approximate solutions (exact ones may require excessive space)
- Maintain lossy summary of Σ via a synopsis data structure (e.g., random sample, hash-based sketch)

Typical data analysis tasks

- Identification of frequent items (or frequent patterns).
- Useful statistics: e.g., frequency moments, quantiles, histograms
- Optimization and graph problems: e.g., clustering, triangle counting

Goal: suitable tradeoffs between accuracy, working memory, update/query time (i.e., *throughput*).

Warm-up:
finding the majority element in a stream

Warm-up: finding the majority element

Majority problem

Given a stream $\Sigma = x_1, x_2, \ldots, x_n$, return the element x (if any) that occurs $> n/2$ times in Σ.

Examples:

- In stream $\Sigma = A, B, A, C, A, D, A, A$, the majority element is A.
- In stream $\Sigma = A, B, C, D, D, E, E$, a majority element does not exist.

Standard off-line setting. Easily solved using linear space in $O(n \log n)$ time, through sorting, or $O(n)$ expected time, through hashing.

Streaming setting. Need 1 or 2 passes, depending on goals

- First pass (Boyer-Moore algorithm): find an element x which is the majority element, if one exists.
- Second pass: check whether x is indeed the majority element.

Boyer-Moore algorithm

The Boyer-Moore algorithm maintains 2 integers *cand* and *count*:

- *cand* contains a candidate majority element (*the true majority element, if one exists*);
- *count* is a counter;

Initialization: $cand \leftarrow$ null and $count \leftarrow 0$.

For each x_t in Σ **do**
  if $count = 0$ **then** $\{cand \leftarrow x_t;\ count \leftarrow 1;\}$
  else $\{$
    if $cand = x_t$ **then** $count \leftarrow count + 1$
    else $count \leftarrow count - 1$
  $\}$

At the end: return *cand*

Example: $\Sigma = A, A, A, C, C, B, B, A, A$

STEP	Cand	Count
1		
2		
3		
4		
5		
6		
7		
8		
9		

Example: $\Sigma = A, A, A, C, C, C, B, B, B$

STEP	Cand	Count
1		
2		
3		
4		
5		
6		
7		
8		
9		

Warm up: finding the majority element

Theorem

Given a stream Σ which contains a majority element m, the Boyer-Moore algorithm returns m using:

- *working memory of size $O(1)$;*
- *1 pass;*
- *$O(1)$ update and query times.*

To check if the returned element is the majority, we need to read the stream again and count its number of occurrences.

Estimating frequencies

Frequency Moments

Consider a stream $\Sigma = x_1, x_2, \ldots, x_n$, whose elements belong to a universe U.

For each $u \in U$ occurring in Σ define its *frequency*

$$f_u = |\{j : x_j = u, 1 \leq j \leq n\}|,$$

i.e., the number of occurrences of u in Σ

Estimating individual frequencies for Σ

OBJECTIVE: in one pass over Σ we want to compute a **small data structure** that enables to derive **estimates** of

- f_u for any given $u \in U$ (*individual frequencies*)

with provable space-accuracy tradeoffs

Observation: the exact computation of all f_u's require space proportional to $|\Sigma|$ using a standard dictionary.

Randomness to the rescue

Randomized algorithm: an algorithm where some decisions depend on random choices.

- Better running time.
- Usually much easier to design and implement (but harder to analyze).

A randimized algorithm has probabilistic behaviors:

- Las Vegas algorithms: the running time is a random variable.
- Montecarlo algorithms: the quality of solution is a random variable.
- We can have both randomized running times and quality of solution.

Randomized algorithms can sometime fail, but we can usually make this probability very small.

Hash functions

Hash function $h : U \to \{0, 1, \ldots, w-1\}$: h is a function that maps each element of U into an integer value in $\{0, 1, \ldots, w-1\}$.

Function h is fixed: an element in U will be always mapped onto the same integer!

However, h is randomly selected from a large set H of hash functions.

The probability that $x \in U$ is mapped on an integer i is over this random selection:

- Uniform probability $\Pr[h(x) = i] = 1/w$: over all possible functions in H, we have probability $1/w$ to select a function mapping x onto i.

Count-min sketch

The first approach we consider is based on the count-min sketch invented by [Cormode,Muthukrishnan 2003].

Main ingredients

- $d \times w$ array C of counters ($O(\log n)$ bits each)
- d hash functions: $h_0, h_1, \ldots, h_{d-1}$, with

$$h_j : U \to \{0, 1, \ldots, w-1\},$$

for every j.

Note that d and w are *design parameters* that regulate the space/time-accuracy tradeoff.

Count-min sketch: algorithm

Initialization: $C[j,k] = 0$, for every $0 \leq j < d$ and $0 \leq k < w$.

For each x_t in Σ **do**

  For $0 \leq j \leq d-1$ **do** $C[j, h_j(x_t)] \leftarrow C[j, h_j(x_t)] + 1$;

At the end of the stream: for any $u \in U$, its frequency f_u can be estimated as:

$$\tilde{f}_u = \min_{0 \leq j \leq d-1} C[j, h_j(u)].$$

Example: $n = 15, d = 3, w = 3$

$\Sigma = A, B, C, B, D, A, C, D, A, B, D, C, A, A, B$

u, f_u	h_0	h_1	h_2
A, 5	0	1	1
B, 4	1	2	1
C, 3	0	0	2
D, 3	1	1	2

Array C		

- $\tilde{f_A} =$
- $\tilde{f_B} =$
- $\tilde{f_C} =$
- $\tilde{f_D} =$

Count-min sketch: analysis

We assume that:

- For each $j \in [0, d-1]$ and each $u, v \in U$ with $u \neq v$, $h_j(u)$ and $h_j(v)$ are independent random variables uniformly distributed in $[0, w-1]$.
- The d hash functions $h_1, h_2, \ldots, h_d$ are mutually independent.

Theorem

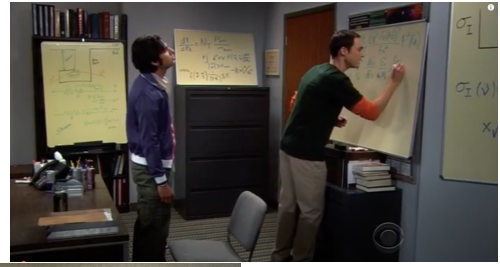
Consider a $d \times w$ count-min sketch for a stream Σ of length n, where $d = \log_2(1/\delta)$ and $w = 2/\epsilon$, for some $\delta, \epsilon \in (0, 1)$. The sketch ensures that for any given $u \in U$ occurring in Σ

$$\tilde{f}_u - f_u \leq \epsilon \cdot n,$$

with probability $\geq 1 - \delta$.

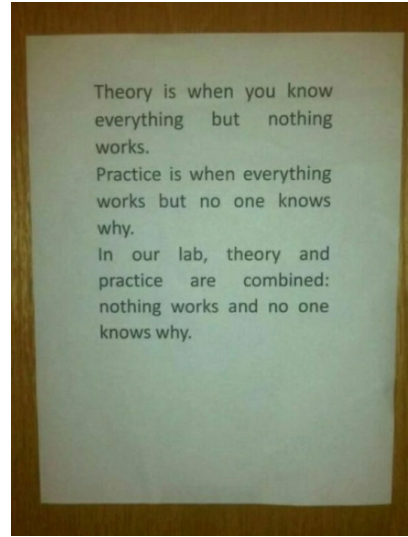
Research in algorithms

Doing research (in algorithm design)



Research in algorithm design

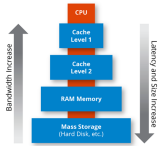
1. Find a computational problem with no known algorithm or with an inefficient algorithm.
2. Design a new algorithm.
3. Provide rigorous guarantees (means "theorems")
4. Implement your algorithm and perform some experiments (means "code").
5. Are you happy about the theoretical/experimental results?
   - No: repeat from step 1;
   - Yes: write paper and move to the next problem.



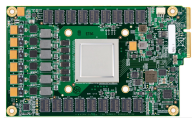
My research

- Hardware-software co-design
- Algorithms for high-dimensional data
- Algorithms with prediction

Hardware-software co-design



Memory hierarchy: it reduced memory bottleneck



Google TPU: fast matrix multiplication in hardware

UPmem memory: a memory with a processor.

Google TPU and UPmem leverage on theoretical ideas developed in '70-'80. Theory matters!

Hardware-software co-design

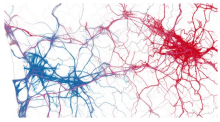
Hardware is continuously evolving to support high computational demand (LLM, big data, ...)

To even reduce resources requirements (less time, less energy, ...), we need a joint hardware-software approach.

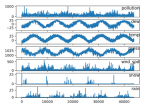
From an algorithmic point of view: we need algorithms that know how to use underline hardware.

Algorithms for high-dimensional data

Graph data from interactions, like in social networks



Trajectories from mobility data



Time series from IoT, industrial processes

```
Token String    Token ID          Embedded Token Vector
     '<s>' ->     0 -> [ 0.1150, -0.1438,  0.0555, ... ]
   '<pad>' ->     1 -> [ 0.1149, -0.1438,  0.0547, ... ]
     '</s>' ->    2 -> [ 0.0018, -0.0922,  0.1025, ... ]
   '<unk>' ->     3 -> [ 0.1149, -0.1439,  0.0548, ... ]
       ',' ->     4 -> [-0.0651, -0.0622, -0.0082, ... ]
     ' the' ->    5 -> [-0.0340,  0.0068, -0.0044, ... ]
       '.' ->     6 -> [ 0.0463, -0.0214, -0.0927, ... ]
      ' to' ->    7 -> [-0.0439,  0.0281,  0.0189, ... ]
     ' and' ->    8 -> [ 0.0523, -0.0288, -0.0254, ... ]
      ' of' ->    9 -> [-0.0732,  0.0070, -0.0286, ... ]
       'a' ->   10 -> [-0.0194,  0.0382, -0.0038, ... ]
                       ...
```

Vectors from transformers in LLMs

Algorithms for high-dimensional data

Big data usually have high-dimensions:

- Black-box in cars record data every second ($>$1K points for just a 20-min ride)
- GPT-4 uses 16K dimensional embeddings

Curse of dimensionality: The running times of standard algorithms are usually *exponential* in the dimension

Randomnized algorithms can break the curse!

Algorithms with predictions

We can construct several predictors using big data and machine learning:

- Predictors exploit hidden properties of data
- Predictions are usually good
- Sometimes, predictions can be very bad!

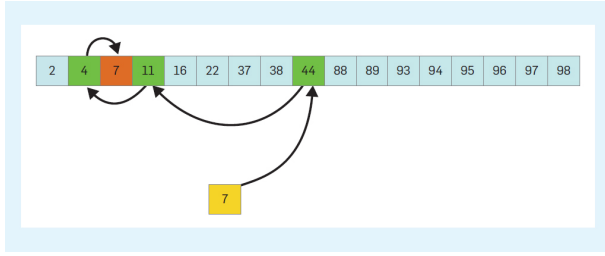
Algorithms with predictions: can we exploit predictors to speed up computations?

- If the prediction is good, we can speed up the computation.
- If the prediction is bad, we can still provide the guarantees provided by standard algorithm.

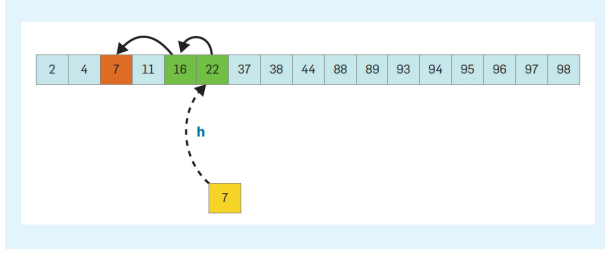
Algorithms with predictions

An example with binary search:

Standard binary search in $O(\log n)$ time:



Binary search with prediction in $O(\log error)$:



Research in algorithms

There are many more open problems in algorithm design.

But first, learn the foundations in algorithms and data structures!