UNIVERSITÄT DUISBURG ESSEN

Universitätsklinikum Essen

technische universität dortmund

fakultät für informatik

Fundamentals of alignment-free sequence analysis: *k*-mer hashing

Part I: Motivation, examples, hash functions

Jens Zentgraf & Sven Rahmann

ACM-BCB 2020

Foundation of most DNA sequence analysis tasks in bioinformatics

1. Read mapping: Find genomic origin(s) of a given DNA sequence (the "read")

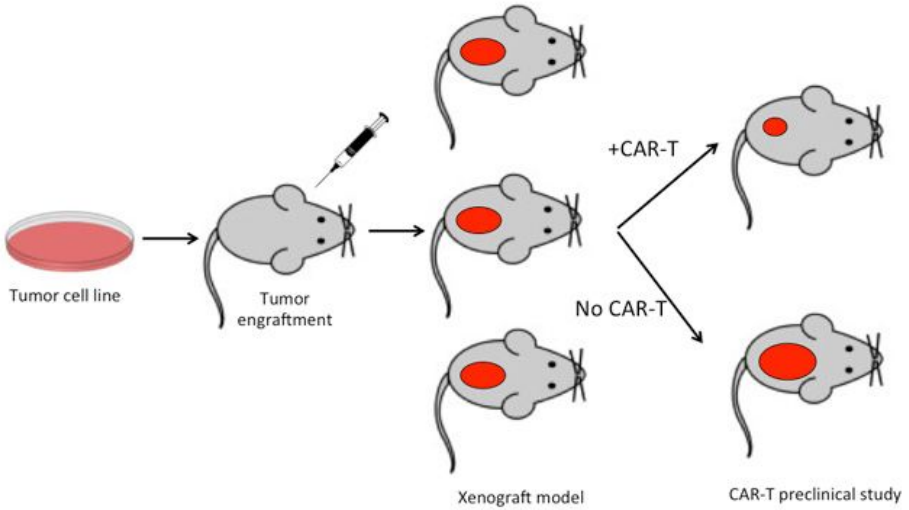
2. Read alignment: Base-by-base comparison of read and genome (often mingled together, but really 2 distinct steps!)

This tutorial: How to short-cut mapping and avoid alignment

- Find all exact occurrences of short *k*-mers
- Do this fast, for billions for *k*-mers

Motivation: Xenograft sorting

(Patient-derived) xenografts

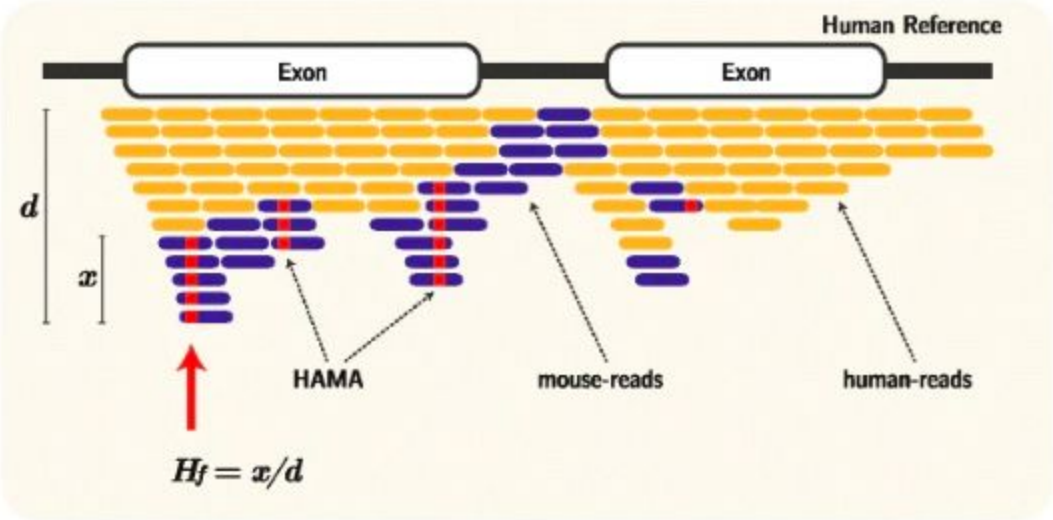


- tumor cell lines
  or patient tumor samples
  implanted in mice
- study tumor heterogeneity,
  evolution
- sequencing of samples
- mixture of human+mouse DNA
- First task: separate/sort reads
  ("xenograft sorting"), or:
  extract graft (human) reads

Source: Creative AniModel,
https://www.creative-animodel.com/Featured-Service/Human-Tumor-Xenograft-Model.html

Problem: Human-Aligned Mouse Alleles (HAMAs)

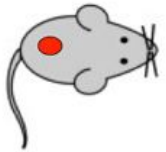
- mouse reads may align to human genome
- may lead to false human (tumor) variant calls
- oncogenes particularly prone to this effect



S. Y. Jo, E. Kim, and S. Kim.
Impact of mouse contamination in genomic profiling of patient-derived models and best practice for robust analysis.
Genome Biology, 20(1):Article 231, Nov 2019.

The xenograft sorting problem

Given: sequenced xenograft sample (reads from two species),
paired-end or single-end,
genomic or transcriptomic reads,



sort the reads into five categories according to species of origin:
host (mouse), graft (human), both, neither, ambiguous

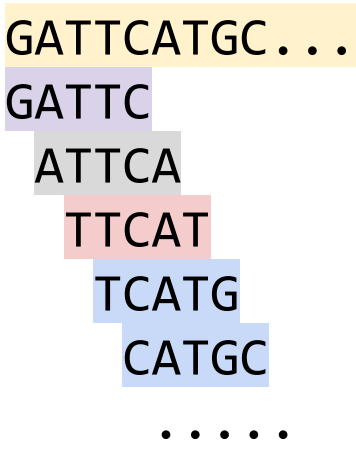
or: partially sort using fewer categories (host, graft, other),
or: count how many reads are in each category,
or: filter (select) only graft (human) reads.

k-mer methods for xenograft sorting

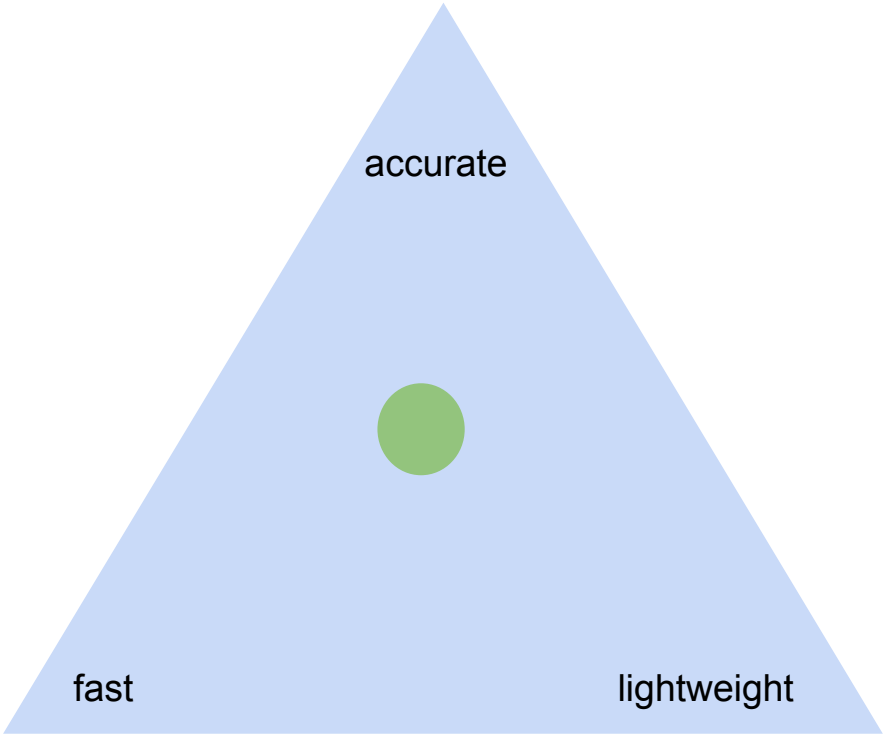
- Partition each read into its *k*-mers

- Look up information on each *k*-mer in hash table
  [*k*-mer ↦ human | mouse | both]
- Absent k-mers occur in neither species.

- Aggregate *k*-mer information
  into a statement about the read
  (majority vote, complex decision rule, …).



Goal: "Fast lightweight accurate xenograft sorting"



fast:

- slow random memory accesses
- 3-way bucketed Cuckoo hashing
- buckets fit within a cache line

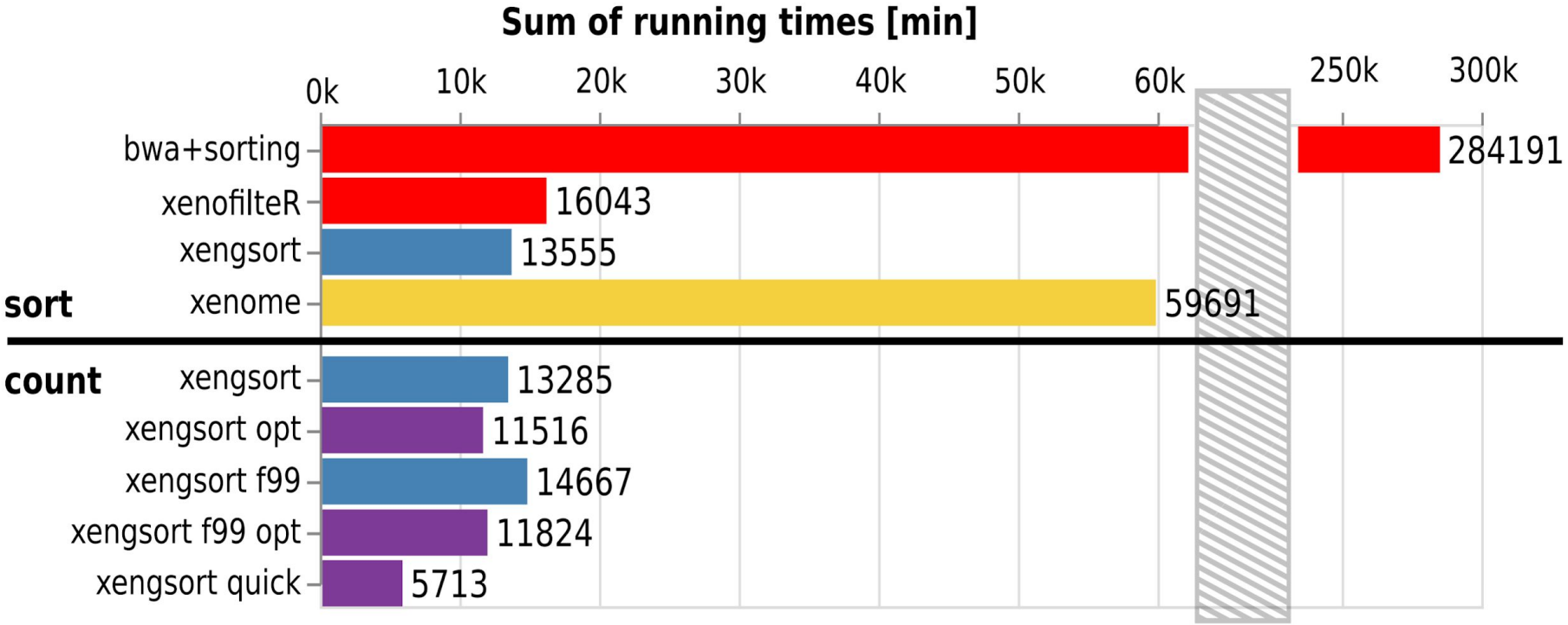
lightweight (small memory footprint):

- 4.5 billion 25-mers + values
- high load (little wasted space)
- quotienting

accurate:

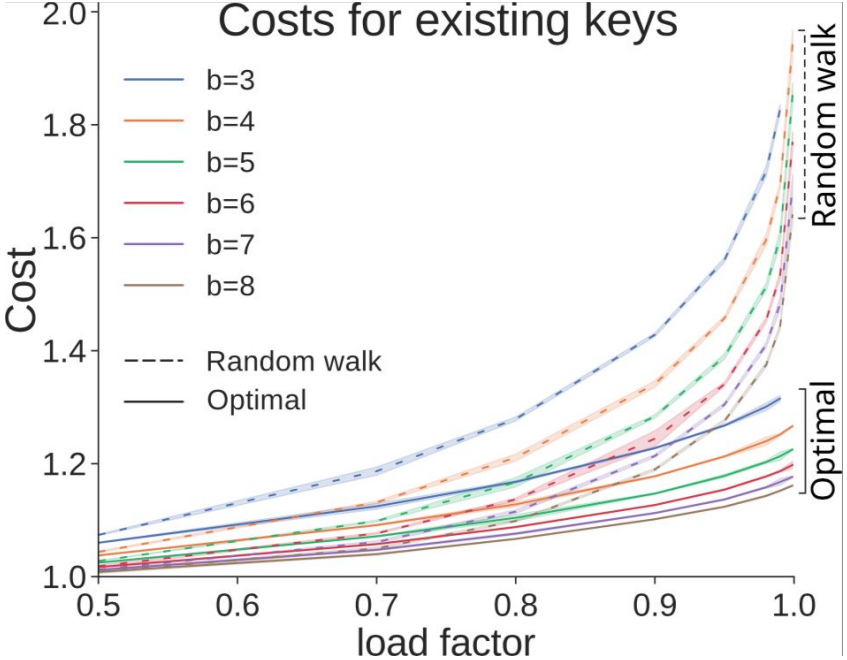
- identical + highly similar sequences
- "weak" *k*-mers
- multi-level decision rule

174 PDX datasets: Running times [CPU minutes]



Zentgraf and R., WABI 2020

Paying attention to the details of hashing is worth it



Look-up costs (#cache misses) for different hash table designs:

- bucketed Cuckoo hashing;
- different bucket sizes ,
- different load factors,
- two insertion strategies.

Examples of alignment-free methods

1. **Xengsort:** Xenograft sorting (already discussed)
   (https://gitlab.com/genomeinformatics/xengsort - 2020)
2. **BCOOL:** Sequencing error correction
   (https://github.com/Malfoy/BCOOL - 2019)
3. **Kraken 2:** metagenomic species identification and quantification
   (https://ccb.jhu.edu/software/kraken2/ - Sep 2019)
4. **Kallisto:** RNA-seq transcript quantification
   (https://pachterlab.github.io/kallisto/ - 2016);
   not for differential expression; use additional tools like sleuth
5. **DE-kupl:** discovery of novel (differentially expressed) transcripts
   (https://transipedia.github.io/dekupl/ - 2017)

k-mers and its encodings

***k*-mer:** any DNA/RNA sequence of length *k*.
There are 4^k different *k*-mers.

Other names: *k*-mer, *q*-gram, *n*-gram, *ℓ*-mer, shingle, ...

k-mer code / encoded *k*-mer: Translating A=0, C=1, G=2, T=3
(or any other bijective map {A,C,G,T} → {0,1,2,3}) for fixed *k*,
a *k*-mer becomes an integer (base-4 number) in {0, 1, …, 4^k-1}.

Example: TATCG ↦ $(30312)_4 = 3 \cdot 256 + 0 \cdot 64 + 3 \cdot 16 + 1 \cdot 4 + 2 \cdot 1 = 822$

Canonical *k*-mers

canonical *k*-mer: DNA is double-stranded;
a *k*-mer ist the same molecule as its reverse complement,
the canonical representation is the **lexicographically smaller** one.
Example: TATCG = CGATA, canonical: CGATA.

canonical code: integer code of canonical *k*-mer
minimum of encodings of *k*-mer and its reverse complement;
always need to evaluate both *k*-mer *x* and rc(*x*).
Example: code(TATCG) = code(CGATA) = min(822, 716) = 716.

Note: works equally well with max() instead of min()

Contiguous vs. gapped *k*-mers

contiguous *k*-mer (standard):

k-mer that occurs as one contiguous substring

gapped / spaced *k*-mer and mask:

- gap pattern given by (symmetric!) mask: e.g.: #__#__#__#
- #: significant positions (*k*) vs. _: gap positions / spacers (*s*)
- *k*-mer by concatenating significant positions (**weight** *k*, **span** *k*+*s*)
- advantage: cover sequence width in fewer steps

```
Example: AGGTCGGTAGGC          AGGTCGGTAGGC      3 k-mers cover
         #__#__#__#  ATGG      ####             12 positions (gapped)
          #__#__#__#  GCTG      ####             6 positions (cont.)
           #__#__#__#  GGAC      ####
```

Key-value stores

General definition:

A key-value store ("key-value database") is a data structure that stores objects or records ("values"), each of which is associated to an immutable "key" object.

Examples:

- Java HashMap
- Python dict
- Databases: redis, Oracle NoSQL, memcached, ...

Restricting values:

Values in key-value databases may be any object, even with different types!
Keys can be any immutable hashable object (often strings or tuples of numbers).

We assume that the value type is known and fixed (value set $V = \{0, .., |V|-1\}$, so values have fixed bit width (e.g. 32 bits).

(Circumvented by storing pointers to arbitrary objects -- what the databases do anyway)

Data structures for key-value-stores (in memory)

Two basic possibilities to look up keys fast:

- **sorting** (binary search)
  - variants of lists (e.g. skip lists)
  - (balanced) search trees
- **hashing** (compute an address / index in an array)
  - typically arrays, but may need to be re-sized
  - collisions must be resolved
- **hybrids** (binning/hashing by prefix, sorted within bin)

Note: on small datasets, do nothing, linear scan is fast enough!

Typical hash functions

Hash functions on DNA (and *k*-mers)

Definitions (hash function f on k-mers for a hash table of size P):

$$f : U \rightarrow \{0, 1, \ldots, P\text{-}1\}$$

- P: table (array) size
- U: universe of all possible keys (here: k-mers for fixed k)
- In concrete applications, f is restricted to actual key set $K \subseteq U$, written $f|_K$
- $f(x) = f(y)$ for $x \neq y$: collision occurs, x and y hash to same location (slot)
- $f|_K$ injective (no collisions on K): perfect hashing (usually when $P >> |K|$)
- $f|_K$ injective and $|K|=P$: minimal perfect hashing.

Encodings (codes) as hash functions ?

Observations:

- k-mer encoding, canonical code,
- any xor-ed (canonical) code with bit mask of $2k$ bits

are already hash functions of DNA k-mers into $\{0, 1, \ldots, 4^k\text{-}1\}$
(perfect hashing!).

However, requires a huge hash table with 4^k slots.
Typically, there are only $|K| = n \ll 4^k$ k-mers in an observed k-mer set K.

Assumption: Hash table size P with $n \leq P \ll 4^k$

Codes mod *P* as hash functions?

Assumption: Hash table size P with $|K| \leq P \ll 4^k$.

Proposal: $f(x)$:= ccode(x) mod P
(remainder of canonical code after division by P)

Properties:

- good: same hash value for x and x's reverse complement
- bad: not flexible (no free parameters)
- bad: may show bias in distribution (non-uniform distribution across slots)

We want close-to-uniform distribution (few collisions),
even if K is an "adversarial" set of k-mers.

Using "standard" hash functions

Idea:

- Take a general-purpose hash function (for bytes/strings) from the internet
- Check that it outputs deterministic 64-bit values
- Take hash value mod *P*

Examples:

- MurmurHash2A (https://en.wikipedia.org/wiki/MurmurHash): 64 bits
- CityHash (google): on byte arrays (like tabulation hashing)
- FarmHash (google): on byte arrays (like tabulation hashing)

Note: Non-cryptographic (i.e easily invertible) hash functions are o.k here!

Tabulation Hashing

Ideas:

- Interpret ($2k$)-bit k-mer as vector of bytes (8-bit units)
  e.g. 23-mer = 46 bits = (almost) 6 bytes
- For each byte i, initialize a random table T_i of $2^8 = 256$ hash values (64 bits)
- Write k-mer $x = (x_0, x_1, \ldots, x_5)$ as 6 bytes
- Compute hash value $f(x) := (T_0[x_0] \oplus T_1[x_1] \oplus \ldots \oplus T_5[x_5]) \bmod P$
- Hash values can have any number of bits (typically 64);
  operation "mod P" is finally applied to obtain range $\{0, \ldots, P\text{-}1\}$.
- Other units than bytes (8 bits) can be used; e.g. 16 bits;
  larger units means much larger (but slightly fewer) tables.
- Tabulation hashing has strong theoretical properties (3-independence).
- **Disadvantage:** Large space requirement (many random numbers, not just 2)

ntHash: specialized DNA hashing

- rolling hash function (like k-mer encoding):
  let x_1, x_2, … be the successive overlapping k-mers
  compute hash value $H(x_i)$ from: $H(x_{i-1})$, removed base, new base
  by updating in constant time instead of re-reading k basepairs.
- special form of tabulation hashing:
  one table with (specially crafted) "random" hash values for each basepair
- Update: $\boldsymbol{H(x_i) = \text{rol}^1(H(x_{i-1})) \oplus \text{rol}^k(h(s[\text{i-1}])) \oplus h(s[i+k\text{-}1])}$
  "Hash value for x_i is hash value of x_{i-1}, rotated left by 1 bit,
  xor-ed with the tabulated value for the outgoing base s[i-1], rotated left by k bits,
  then xor-ed with the tabulated value for the incoming base $s[i+k\text{-}1]$ as is."

Hamid Mohamadi, Justin Chu, Benjamin P. Vandervalk, Inanc Birol, ntHash: recursive nucleotide hashing.
Bioinformatics, Volume 32, Issue 22, 15 November 2016, Pages 3492–3494. (https://doi.org/10.1093/bioinformatics/btw397)

Randomized Rotate-Multiply-Offset

Proposal (bit rotation, randomization): Pick two integers

- multiplier *a* odd in {1, 3, …, 4^k-1},
- offset *b* in {0, 1, 2, …, 4^k-1};

$$f(x) := [(a \ \mathrm{rot}_k(\mathrm{ccode}(x)) \oplus b) \bmod 4^k \,] \bmod P$$

  - rot_k: cyclic rotation by *k* bits: inner bits outside, outer bits inside.

Good practical properties:

- same hash value for x and x's reverse complement
- The part in [...] is a random **bijection** on the universe *U* (if |*U*| is a power of 2)
- If biased, just pick different random *a*, *b*.

Saving space with quotienting

Keys: canonical codes of 25-mers (50 bits)
Values: species (5 classes: 3 bits)

4.5 billion k-mers: reference genomes, alternative alleles, cDNA transcripts:
53 bits per entry, load 0.88: **33.88 GB** for hash table 😩

Quotienting to the rescue:

- Do not store full keys (k-mers), but only "quotients" (here 20 bits),
  plus hash function choice (2 bits) plus values (3 bits) → 25 bits per entry:

15.98 GB for hash table 😀
(could be slightly reduced by higher load, value compression, etc.)

Quotienting: Details

Keys are encoded canonical k-mers (half of set $[4^k] := \{0, .., 4^k\text{-}1\}$).

Step 1: Bijective randomizing function $[4^k] \to [4^k]$ with a odd

$$g_{a,b}(x) := [a \cdot (\mathrm{rot}_k(x) \text{ xor } b)] \bmod 4^k$$

Step 2: Map to buckets (simply mod p: number of buckets). Define

$$f(x) := g_{a,b}(x) \bmod p \quad \text{and} \quad q(x) := g_{a,b}(x) \mathbin{//} p \ .$$

Then x can be uniquely reconstructed
from $f(x)$ ("hash value, "bucket number") and $q(x)$ ("fingerprint", "quotient").
Sufficient to store $q(x)$ in bucket $f(x)$ (and which hash function was chosen).