Zentgraf and Rahmann *Algorithms Mol Biol* (2021) 16:2
https://doi.org/10.1186/s13015-021-00181-w

Algorithms for Molecular Biology

RESEARCH Open Access

Fast lightweight accurate xenograft sorting

Jens Zentgraf[1] and Sven Rahmann[1,2*]

Abstract

Motivation: With an increasing number of patient-derived xenograft (PDX) models being created and subsequently sequenced to study tumor heterogeneity and to guide therapy decisions, there is a similarly increasing need for methods to separate reads originating from the graft (human) tumor and reads originating from the host species' (mouse) surrounding tissue. Two kinds of methods are in use: On the one hand, alignment-based tools require that reads are mapped and aligned (by an external mapper/aligner) to the host and graft genomes separately first; the tool itself then processes the resulting alignments and quality metrics (typically BAM files) to assign each read or read pair. On the other hand, alignment-free tools work directly on the raw read data (typically FASTQ files). Recent studies compare different approaches and tools, with varying results.

Results: We show that alignment-free methods for xenograft sorting are superior concerning CPU time usage and equivalent in accuracy. We improve upon the state of the art sorting by presenting a fast lightweight approach based on three-way bucketed quotiented Cuckoo hashing. Our hash table requires memory comparable to an FM index typically used for read alignment and less than other alignment-free approaches. It allows extremely fast lookups and uses less CPU time than other alignment-free methods and alignment-based methods at similar accuracy. Several engineering steps (e.g., shortcuts for unsuccessful lookups, software prefetching) improve the performance even further.

Availability: Our software *xengsort* is available under the MIT license at http://gitlab.com/genomeinformatics/xengsort. It is written in numba-compiled Python and comes with sample Snakemake workflows for hash table construction and dataset processing.

Keywords: Xenograft sorting, Alignment-free method, Cuckoo hashing, k-mer

Introduction

To learn about tumor heterogeneity and tumor progression under realistic *in vivo* conditions, but without putting human life at risk, one can implant human tumor tissue into a mouse and study its evolution. This is called a (patient-derived) xenograft (PDX). Over time, several samples of the (graft/human) tumor and surrounding (host/mouse) tissue are taken and subjected to exome or whole genome sequencing in order to monitor the changing genomic features of the tumor. This information can be used to predict the response to different chemotherapy alternatives and to monitor treatment success or failure. A key step in such analyses is *xenograft sorting*, i.e., separating the human tumor reads from the mouse reads. A recent study [1] showed that if such a step is omitted, several mouse reads would be aligned to certain regions of the human genome (HAMA: human-aligned mouse allele) and induce false positive variant calls for the tumor; this especially concerns certain oncogenes.

Several tools have been developed for xenograft sorting, motivated by different goals and using different approaches; a summary appears below. Here we improve upon the existing approaches in several ways: by using carefully engineered *k*-mer hash tables, our approach is both *faster* and needs *less memory* than existing tools. By

*Correspondence: Sven.Rahmann@uni-due.de
[2] Genome Informatics, Institute of Human Genetics, University Hospital Essen, University of Duisburg-Essen, Essen, Germany
Full list of author information is available at the end of the article

Table 1 Tools for xenograft sorting and read filtering with key properties

Tool	Ref.	Input	Operations	Language
XenofilteR	[2]	Aligned BAM	Filter	R
Xenosplit	[3]	Aligned BAM	Filter, count	Python
Bamcmp	[4]	Aligned BAM	Partial sort	C++
Disambiguate	[5]	Aligned BAM	Partial sort	Python or C++
BBsplit	[6]	Raw FASTQ	Partial sort	Java
Xenome	[7]	Raw FASTQ	Count, sort	C++
Xengsort	(This)	Raw FASTQ	Count, sort	Python + numba

See text for definition of operations

designing a new decision function, we also obtain *fewer unclassified reads* and in some cases even *higher classification accuracy*. Since we use a comprehensive reference of the genome and transcriptome, we are in principle able to process genome, exome, and transcriptome samples of xenografts. Of course, different sources may exhibit different error distributions and require distinct optimized parameter sets for classification. Nevertheless, our evaluation shows that we obtain good results on all of exomes, genomes and transcriptomes with the same parameter set.

Concerning related work, we distinguish alignment-based methods that work on already aligned reads (BAM files), versus alignment-free methods that directly work on short subsequences (k-mers) of the raw reads (FASTQ files).

Alignment-based methods scan existing alignments in BAM files and test whether each read maps better to the graft genome or to the host genome. Differences result from different parameter settings used for the alignment tool (often `bwa` or `bowtie2`) and from the way "better alignment" is defined by each of these tools. Alignment-free methods use a large lookup table to associate species information with each k-mer.

In Table 1, we list properties of existing tools and of *xengsort*, our implementation of the method we describe in this article. These tools support different operations: Operation "count" outputs proportions of reads belonging to each category (host, graft, etc.); operation "sort" sorts reads or alignments into different files according to origin, ideally into five categories: host, graft, both, neither, ambiguous; a "partial sort" only has three categories: host, graft, both/other; operation "filter" writes only an output file with graft reads or alignments. The sort operation is more general than the filter or partial sort operation and allows full flexibility in downstream processing. The count operation, when it is available separately, is faster than counting the output of the sort operation, because it avoids the overhead of creating output files.

XenofilteR, *Xenosplit*, *Bamcmp* and *Disambiguate* all work on aligned BAM files. This means that the reads must be mapped and aligned with a supported read mapper first (typically, 'bwa mem') and the resulting BAM file must be sorted in a specific way required by the tool. The tool is typically a script that reads and compares the mapping scores and qualities in the two BAM files containing host and graft alignments. In principle, all of these tools do the same thing; large differences result rather from different alignment parameters than the tool itself. We therefore picked *XenofilteR* as a representative of this family, also because it performed well in a recent comparison [1].

BBsplit (part of BBTools) is special in the sense that it performs the read mapping itself, against multiple references simultaneously, based on k-mer seeds. Unfortunately, only up to approximately 1.9 billion k-mers can be indexed because of Java's array indexing limitations (up to 2^{31} elements) and a table load limit of 90%; so *BBsplit* was not usable for our human-mouse index that contains approximately $4.5 \cdot 10^9 > 2^{32}$ k-mers.

The tool *xenome* [7] is similar to our approach: It is based on a large hash table of k-mers and sorts the reads into several categories (host, graft, both, neither, ambiguous). A read is classified based on its k-mer content according to relatively strict rules. We found the threading code of *xenome* to be buggy, such that the pure counting mode resulted in a deadlock and produced no output. The sorting mode produced the complete output but then did not terminate either.

Recent studies [1, 8, 9] have compared the computational efficiency of several methods, as well as the classification accuracy of these methods and the effects on subsequent variant calling after running vs. not running xenograft sorting. The results were contradictory, with some studies reporting that alignment-based tools are more efficient than alignment-free tools, and different tools achieving highest accuracy. Our interpretation of the results of [1] is that each of the existing approaches is able to sort with good accuracy and the main difference is in computational efficiency. Results about efficiency have to be interpreted with care because sometimes the time for alignment is included and sometimes not.

Methods

Overview

By considering all available host and graft reference sequences (both transcripts and genomic sequences of mouse and human), we build a large key-value store that allows us to look up the species of origin (host, graft or both) of each DNA/RNA k-mer that occurs in either species. A sequenced dataset (a collection of single-end or

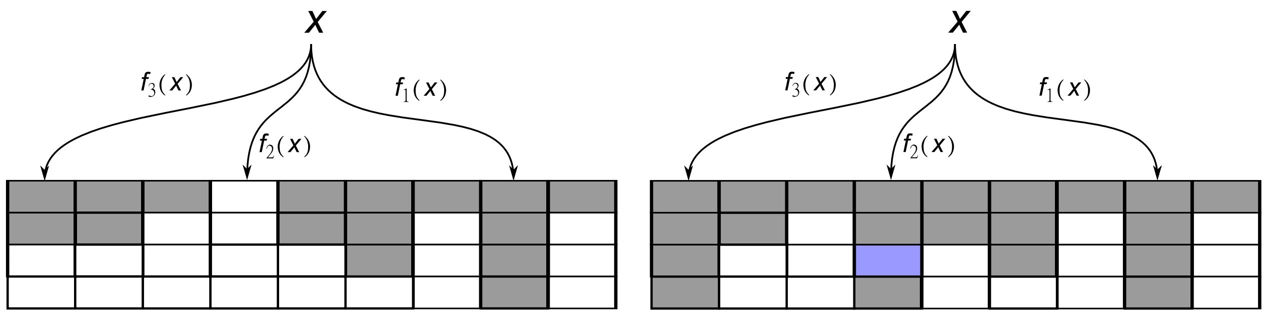

Fig. 1 Illustration of (3,4) Cuckoo hashing with 3 hash functions and buckets of size 4. Left: Each key-value pair can be stored at one of up to 12 locations in 3 buckets. For key x, the bucket given by $f_1(x)$ is full, so bucket $f_2(x)$ is attempted, where a free slot is found. Right: If all hb slots are full, key x is placed into one of these slots at random (blue), and the currently present key-value pair is evicted and re-inserted into an alternative slot

paired-end FASTQ files) is then processed by iterating over reads or read pairs, looking up the species of origin of each k-mer in a read (host, graft, both or none) and classifying the read based on a decision rule.

Our implementation of the key-value store as a three-way bucketed Cuckoo hash table makes k-mer lookup faster than in other methods; the associated value can often be retrieved with a single random memory access. A high load factor of the hash table, combined with the technique of quotienting, ensures a low memory footprint, without resorting to approximate membership data structures, such as Bloom filters.

Key-value stores of canonical k-mers

We partition the reference genomes (plus alternative alleles and unplaced contigs) and transcriptomes into short substrings of a given length k (so-called k-mers); we evaluated $k \in \{23, 25, 27\}$. For each k-mer ("key") in any of the reference sequences, we store whether it occurs exclusively in the host reference, exclusively in the graft reference, or in both, represented by "values" 1, 2, 3, respectively. For the host- and graft-exclusive k-mers, we also store whether a closely similar k-mer (at Hamming distance 1) occurs in the other species (add value 4); such a k-mer is then called a *weak* (host or graft) k-mer. This idea extends the k-mer classification of *xenome* [7], where a k-mer can be host, graft, both, or marginal, the latter category comprising both our weak host and weak graft k-mers. So we store, for each k-mer, a value from the 5-element set "host" (1), "graft" (2), "both" (3), "weak host" (5), "weak graft" (6). This value is stored using 3 bits. While a more compact base-5 representation is possible (e.g., storing 3 values with $125 < 128 = 2^7$ combinations in 7 bits instead of in 9 bits), we decided to use slightly more memory for higher speed.

To be precise, we do not work on k-mers directly, but on their canonical integer representations (*canonical codes*), such that a k-mer and its reverse complement map to the same number. We use a simple base-4 numeric encoding $A \mapsto 0$, $C \mapsto 1$, $G \mapsto 2$, $T/U \mapsto 3$, e.g., reading the 4-mer AGCG as $(0212)_4 = 38$ and its reverse complement CGCT as $(1213)_4 = 103$. The *canonical code* is then the *maximum* of these two numbers; here the canonical code of both AGCG and CGCT is thus 103. (In *xenome*, canonical k-mer codes are implemented with a more complex but still deterministic function of the two base-4 encodings; in other tools, it is often the minimum of the two encodings.) For odd k, there are exactly $c(k) := 4^k/2$ different canonical k-mer codes, so each can be stored in $2k - 1$ bits in principle. However, implementing a *fast* bijection of the set of canonical codes (which is a subset of size $c(k)$ of $\{0 .. (4^k - 1)\}$) to $\{0 .. (c(k) - 1)\}$ seems difficult, so we use $2k$ bits to store the canonical code directly, which allows faster access. However, we use quotienting, described below, to reduce the size of the stored k-mer code.

Multi-way bucketed quotiented Cuckoo hashing

We use multi-way bucketed Cuckoo hash table as the data structure for the k-mer key-value store. Let C be the set of canonical codes of k-mers; as explained above, we take $C = \{0 .. (4^k - 1)\}$, even though only half of the codes are used (for odd k). Let P be the set of locations (buckets) in the hash table and p their number; we set $P := \{0 .. (p - 1)\}$. Each key can be stored at up to h different locations (buckets) in the table. The possible buckets for a code are computed by h different hash functions $f_1, f_2, \ldots, f_h : C \to P$. Each bucket can store up to a certain number b of key-value pairs. So there is space for $N := pb$ key-value pairs in the table overall, and each pair can be stored at one of hb locations in h buckets. Together with an insertion strategy as described below, this framework is referred to as (h, b) Cuckoo hashing. Classical Cuckoo hashing uses $h = 2$ and $b = 1$; for this work, we use $h = 3$ and $b = 4$. A visualization is provided in Fig. 1. Using several hash functions and larger buckets increases the load limit; using $h = 3$ and $b = 4$ allows a load factor of over 99.9% [10, Table 1], while classical Cuckoo hashing only allows to fill 50% of the table.

Search

Searching for a key-value pair works as follows. Given key (canonical code) x, first $f_1(x)$ is computed, and this bucket is searched for key x and the associated value. If it is not found, buckets $f_2(x)$ and then $f_3(x)$ are searched similarly. Each bucket access is a random memory lookup and most likely triggers a cache miss. We can ensure that each bucket is contained within a single cache line (by using additional padding bits if necessary). Then, the number of cache misses is limited to $h = 3$ for one search operation.

When we fill the table well below the load limit (at 88% of 99.9%), we are able to store many key-value pairs in the bucket indicated by the *first* hash function f_1, and only incur a single cache miss when looking for them. Unsuccessful searches (for k-mers that are not present in either host or graft genome) need all h memory accesses. However, optimizations are possible and described below (see "Performance engineering" section).

Insert

Insertion of a key-value pair works as follows. First, the key is searched as described above. If it is found, the value is updated with the new value. For example, if an existing host k-mer is to be inserted again as a graft k-mer, the value is updated to "both". If the key is not found, we check whether any of the buckets $f_1(x)$,$f_2(x)$,$f_3(x)$ (in that order) contains a free slot. If this is the case, x and its value are inserted there. If all buckets are full, a random slot among the hb slots is picked, and the key-value pair stored there is evicted (like a cuckoo removes eggs from other birds' nests) to make room for x and its value. Then an alternative location for the evicted element is searched. This process may continue for several iterations and is called a "random walk" through the table. If the walk becomes too long (longer than 5000 steps, say), we declare that the table is too full, and construction fails and has to be restarted with a larger table or different random seed.

Our implementation requires that the size (number of buckets p) of the hash table is known in advance, so we can pre-allocate it. The genome length is a good (over-) estimate of the number of distinct k-mers and can be used. We recently presented a practical algorithm [11] to optimize the assignment of k-mers to buckets (i.e., their hash function choices) such that the average search cost of present k-mers is minimized to the provable optimum. This optimization takes significant additional time and requires large additional data structures; so we took the opportunity here to evaluate whether it significantly improves lookup times in comparison to a table filled by the above random walk strategy (see "Results").

Bijective hash functions and quotienting

In principle, we need to store the $2k$ bits for the canonical k-mer code x and the 3 bits for the value at each slot. However, by using hash functions of the form $f(x) := g(x) \bmod p$, where p is the number of buckets and g is a *bijective* (randomized) transformation on the full key set $\{0 .. (4^k - 1)\}$, we can encode part of x in $f(x)$: Note that from $f(x)$ and $q(x) := g(x)//p$ (integer division), we can recover $g(x) = p \cdot q(x) + f(x)$, and since g is bijective, we can recover x itself. This means that we only need to store $q(x)$, not x itself in bucket $f(x)$, which only takes $\lceil 2k - \log_2 p \rceil$ instead of $2k$ bits. However, since we have h alternative hash functions, we also need to store *which* hash function we used, using 2 bits for $h = 3$ (0 indicating that the slot is empty). This technique is known as *quotienting*. It gives higher savings for smaller buckets (for constant $N = pb$, smaller b means larger p), but on the other hand the load limit is smaller for small b. We find $b = 4$ to be a good compromise, allowing table loads of 99.9%.

For the bijective part $g(x)$, we use affine functions of the form

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

where rot_k performs a cyclic rotation of k bits (half the width of x), moving the "random" inner bits to outer positions and the less random outer bits (due to the max operation when taking canonical codes) inside, b is a $2k$-bit offset, and a is an odd multiplier. Picking a "random" hash function means picking random values for a and b.

Lemma 1 *For any $2k$-bit number b and any odd $2k$-bit number a, the function $g_{a,b}$ is a bijection on $K := \{0 .. (4^k - 1)\}$, and its inverse can be efficiently obtained.*

Proof Let $y = g_{a,b}(x)$. By definition, the range of $g_{a,b}$ on K is a subset of K. Because $|K|$ is a power of 2 and a is odd, the greatest common divisor of $|K|$ and a is 1, and so there exists a unique multiplicative inverse a' of a modulo $4^k = |K|$, such that $aa' = 1 \pmod{4^k}$. This inverse a' can be obtained efficiently using the extended Euclidean algorithm. The other operations (xor b, rot_k) are inverses of themselves; so we recover $x = \mathrm{rot}_k([(a' \cdot y) \bmod 4^k] \text{ xor } b)$.□

In summary, each stored canonical k-mer needs $2 + 3 + \lceil 2k - \log_2 p \rceil$ bits to remember the hash function choice and to store the value (species) and the quotient, respectively. For $k=25$ and $p=1\,276\,595\,745$ buckets, this amounts to 25 bits per k-mer, or 100 bits for each bucket of 4 k-mers. To ensure cache line (512 bits) aligned buckets, we could use 500 bits for 5 buckets and insert 12 padding bits; however, we chose to use less memory and let a few buckets cross cache line boundaries, accepting the resulting speed decrease.

Performance engineering

Software prefetching

Prefetching refers to instructing the memory system to fetch data from RAM into the cache hierarchy before the CPU actually needs the data. This can reduce the time spent by the CPU waiting for data, especially in lookup-intensive applications such as this one. Hardware prefetching is automatically performed by the CPU based on memory access patterns (i.e., a linear scan over a large array). Software prefetching refers to application-controlled prefetching. Our application *xengsort* supports three levels of prefetching: none (0), prefetching the second choice bucket before searching the first choice bucket (1), or prefetching both second and third choice buckets before searching the first choice bucket (2). The disadvantage is that, if the search of the first bucket is successful, the memory system has done unnecessary work, possibly slowing down other threads that want to access different memory locations at the same time. As a consequence, software prefetching should only be enabled if the second and/or third bucket must be examined frequently, i.e., at high load factors, or when many unsuccessful lookups can be expected.

Shortcuts for unsuccessful lookups

As described so far, unsuccessful lookups are slow because all three buckets must be completely examined, even though software prefetching may solve part of the problem. In addition, algorithmic optimizations are possible, with 0 to 2 extra bits of memory per bucket.

The following shortcut is possible without using any additional memory: If, say, the first bucket $f_1(x)$ contains an empty slot, we do not need to search further, because the random walk insertion procedure produces a tight layout, in the sense that if a single element could have been moved to an "earlier" bucket, it would have been done.

Using a single additional "shortcut" flag bit per bucket, we can store whether there exists any element in a "later" choice bucket that could have been inserted into this bucket, had there been more space. So a set bit (value 1) indicates that later choices must be searched if the element is not found in this bucket, while a cleared bit (value 0) indicates that a search can be terminated unsuccessfully when the element is not found in this bucket. The same idea has been proposed by Alain Espinosa as "unlucky buckets trace" [12].

Using a second bit per bucket, the resolution of this type of information can be further improved: One bit indicates that there exists an element whose first choice would have been this bucket, but that is stored at its *second choice* bucket. The other bit indicates that there exists an element whose first or second choice would have been this bucket, but that is stored at its *third choice* bucket. If the element is not found in the current bucket, then, depending on the bit combination, the search can be stopped early (0,0), only the second choice needs to be checked (1,0), only the third choice needs to be checked (0,1), or both (1,1).

These shortcuts work best if (a) there are many unsuccessful lookups and (b) they are evaluated only after the search in the first bucket was unsuccessful. The performance gains are evaluated in the "Results" section.

No key deletions

In principle, our implementation of Cuckoo hashes allows for easy deletion of keys: in the corresponding slot, simply set the hash choice bits to zero. However, this will destroy the tight layout mentioned in the previous paragraph and invalidate the shortcut flag bits; therefore subsequent unsuccessful lookups must examine all locations and become more expensive. Restoring the tight layout may involve many iterations of moving keys throughout the table, and hence is an expensive operation. As our application never needs to delete an existing key, we fully benefit from the above mentioned shortcuts.

Additional memory savings

Theoretically, the two bits indicating the hash function choice for each slot may be saved by using separate tables for each hash function. However, this drastically decreases the load limits and overall results in higher memory requirements. A better alternative is to exploit that the order of slots in a bucket is arbitrary, so we may enforce a fixed order: first all keys that are present with their final hash function, then in order all keys resulting from earlier hash functions, and finally all empty slots. Thus the configuration of a bucket is given by a non-negative $(h+1)$-tuple $(c_h,\ldots,c_1,c_0)\geq 0$ with sum b, where c_i is the number of elements in the bucket which are present because of their i-th hash function (for $i\geq 1$), and c_0 is the number of empty slots. Especially for large b, there are much fewer possible such tuples than $(h+1)^b$. For the case of $(h,b)=(3,4)$, there are 35 such tuples, and the configuration can be encoded in 6 instead of 8 bits. Encoding more than one bucket jointly results in further savings. Similarly, the values for a bucket can be

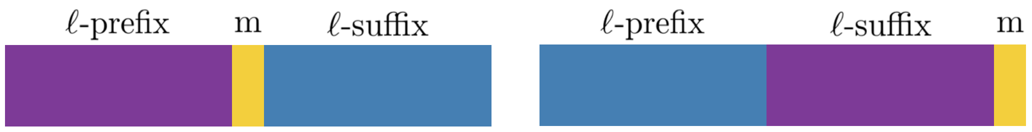

Fig. 2 A *k*-mer is partitioned into its ℓ-prefix, a middle base and its ℓ-suffix. Efficient local re-sorting of *k*-mers according to common ℓ-prefix and ℓ-suffix yields groups of *k*-mers that differ only in their middle base

encoded jointly. For example, given a value set of size 5, where a value requires 3 bits, there are $5^4 = 625$ different value combinations in a bucket, which can be encoded in 10 bits instead of $4 \cdot 3 = 12$. In a practical setting (human/mouse, $k = 25$, load 0.88; see Table 3), combining both options reduces the hash table size by 0.5 GB from 15.9 GB to 15.4 GB. However, these savings come at the cost of increased CPU time for decoding the configuration or values. Neither option has been implemented yet, but will be added in a future release.

Annotating weak *k*-mers

A *k*-mer that occurs only in the host (graft) reference, but has a Hamming-distance-1 neighbor in the graft (host) reference, is called a *weak* host (graft) *k*-mer. So for a weak *k*-mer, a single nucleotide variation could flip its assigned species, while a *k*-mer that is not weak is more robust in this sense. After the hash table has been constructed with all *k*-mers and their values "host", "graft" or "both", we mark weak *k*-mers by modifying the value, setting an additional "weak" bit. In principle, we could scan over the *k*-mers and query all 3*k* neighbors of each *k*-mer, but this is inefficient.

Instead, we extract from the hash table a complete list *L* of *k*-mers and their reverse complements (not canonical codes; approx. $9 \cdot 10^9$ entries for $4.5 \cdot 10^9$ distinct *k*-mers), together with their current values. To save memory, this list is created and processed in 16 chunks according to the first two nucleotides of the *k*-mer, thus needing approx. 4.5 GB of additional memory temporarily. Since we use odd $k = 2\ell + 1$, we can partition a *k*-mer into its ℓ-prefix, its middle base and its ℓ-suffix (Fig. 2). We make use of the following observation.

Observation 1 For $k = 2\ell + 1$, two k-mers x, y with Hamming distance 1 differ either in their ℓ-suffix, in the ℓ-suffix of their reverse complement or in their middle base.

Consider first the case where the difference is in the ℓ-suffix. We thus partition the sorted chunk into blocks of constant $(\ell + 1)$-prefixes. Different blocks are processed independently in parallel threads. The ℓ-suffixes of all pairs of *k*-mers in such a block are queried with a fast bit-vector test for Hamming distance 1. If a pair is found and the *k*-mers occur in different species, the "weak bit" (value 4) is set on both *k*-mers of the pair. Now consider the case where the difference is in the ℓ-prefix. Because reverse complements are included in the full list, this case is already covered.

It remains to find pairs of *k*-mers that differ only in their middle base. We thus conceptually re-partition the chunk into blocks of constant ℓ-prefixes. We now switch the order of ℓ-suffix and middle base (Fig. 2) and re-sort each block internally. This is a cache-friendly local operation on typically relatively small blocks. Now Hamming-distance-1 groups that differ in their original middle base occur consecutively in a block and agree in their 2ℓ-prefix. A scan over the block reveals all relevant pairs.

Finally, the updated values are transferred to the values of the canonical *k*-mers in the hash table.

Reference sequences

To build the *k*-mer hash table of the human (GRCh38, hg38) and mouse (GRCm38, mm10) genome and transcriptome, we obtained the "toplevel DNA" genome FASTA files, which include both the primary assembly, unplaced contigs and alternative alleles, and the "all cDNA" files, which contain the known transcripts, from the ensembl FTP site, release 98.

As the alternative alleles of the human and mouse toplevel references contain mostly Ns to keep positional alignment of alternative alleles to the consensus reference, they decompress to huge FASTA files (over 60 GB for human, over 12 GB for mouse). Therefore we condensed the toplevel reference sequences by replacing runs of more than 25 Ns by 25 Ns. This does not change the *k*-mer content, as *k*-mers containing even a single N are ignored. It does provide an efficiency boost to alignment-based tools because read mappers build an index of every position in the genome and typically replace runs of Ns by random sequence.

Fragment classification

Given a sequenced fragment (single read or read pair), we query each *k*-mer of the fragment about its origins; *k*-mers with undetermined bases (Ns) are ignored. Our implementation reads large chunks (several MB) of FASTQ files and distributes read classification over several threads (we found that 8 threads saturate the I/O).

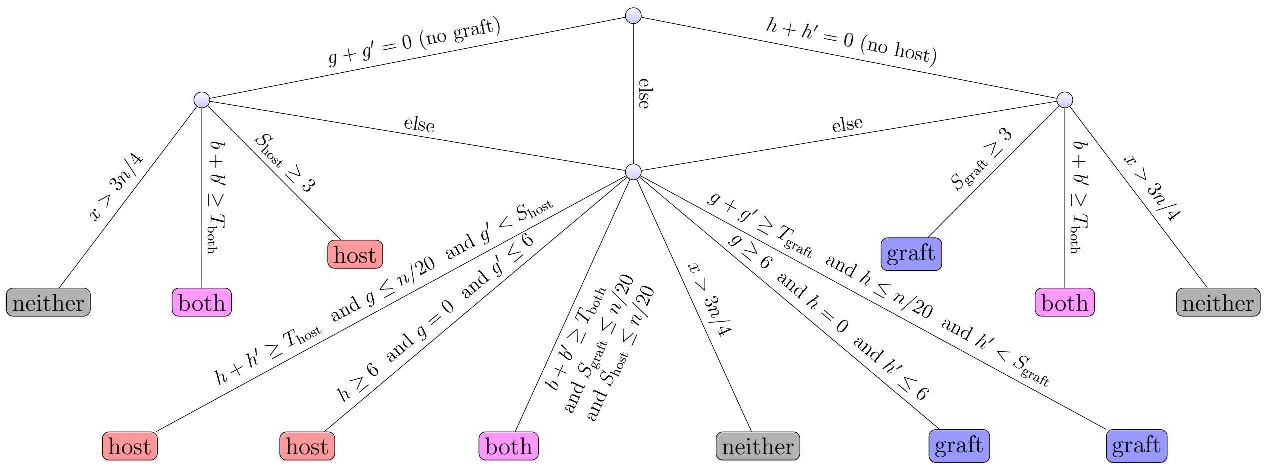
$g + g' = 0$ (no graft)
$h + h' = 0$ (no host)
else
else
else
$x > 3n/4$
$b + b' \geq T_{\text{both}}$
$S_{\text{host}} \geq 3$
neither
both
host
$S_{\text{graft}} \geq 3$
$b + b' \geq T_{\text{both}}$
$x > 3n/4$
graft
both
neither
$h + h' \geq T_{\text{host}}$ and $g \leq n/20$ and $g' < S_{\text{host}}$
$h \geq 6$ and $g = 0$ and $g' \leq 6$
$b + b' \geq T_{\text{both}}$ and $S_{\text{graft}} \leq n/20$ and $S_{\text{host}} \leq n/20$
$x > 3n/4$
$g \geq 6$ and $h = 0$ and $h' \leq 6$
$g + g' \geq T_{\text{graft}}$ and $h \leq n/20$ and $h' < S_{\text{graft}}$
host
host
both
neither
graft
graft

Fig. 3 Decision rule tree for classifying a DNA fragment from *k*-mer statistics $(h, h', g, g', b, x; n)$, meaning number of *k*-mers of type "host" (h), "weak host" (h'), "graft" (g), "weak graft" (g'), "both" (b), and number of *k*-mers not present in the key-value store (x), respectively; n is the total number of (valid) *k*-mers in the fragment. We also use weighted scores $S_{\text{host}} := h + \lfloor h'/2 \rfloor$ and $S_{\text{graft}} := g + \lfloor g'/2 \rfloor$ and thresholds $T_{\text{host}} := \lfloor n/4 \rfloor$, $T_{\text{graft}} := \lfloor n/4 \rfloor$ and $T_{\text{both}} := \lfloor n/4 \rfloor$. A fragment is thus classified as "host", "graft", "both", "neither", or "ambiguous". Category "ambiguous" is chosen if no other rule applies and no "else" rule is present in a node

We collect *k*-mer statistics for each fragment (adding the numbers of both reads for a read pair): let n be the number of (valid) *k*-mers in the fragment. Let h be the number of (non-weak) host *k*-mers and h' the number of weak host *k*-mers, and analogously define g and g' for the graft species. Further, let b be the number of *k*-mers occuring in both species, and let x be the number of *k*-mers that were not found in either species.

Based on the vector $(h, h', g, g', b, x; n)$, we use a tree of hierarchical rules to classify the fragment into one of five categories: "host", "graft", "both", "neither" and "ambiguous". Categories "host" and "graft" are for reads that can be clearly assigned to one of the species. Category "both" is for reads that match equally well to both references. Category "neither" is for reads that contain many *k*-mers that cannot be found in the key-value store; these could point to technical problems (primer dimers) or contamination of the sample with other species. Finally, category "ambiguous" is for reads that provide conflicting information. Such reads should not usually be seen; they could result from PCR hybrids between host and graft during library preparation.

The precise rules are shown in Fig. 3. The rules are designed to arrive at easy decisions quickly. For example, at the root node, if there are no graft *k*-mers at all $(g + g' = 0)$, then an easy decision can be made between the classes "host" (if there is at least a little evidence of host *k*-mers, i.e., $S_{\text{host}} \geq 3$, where $S_{\text{host}} := h + \lfloor h'/2 \rfloor$), "both" (if there are sufficiently many such *k*-mers, i.e. $b \geq T_{\text{both}} := \lfloor n/4 \rfloor$, but the "host" class does not apply), and "neither" (if there are sufficiently many such *k*-mers, i.e. $x \geq 3n/4$). If none of these conditions is true, the "ambiguous" class is chosen. A symmetric quick decision rule exists for the case that no host *k*-mers exist $(h + h' = 0)$. If no quick decision can be made, more complex rules are applied: The next test is whether there are no (strong) graft *k*-mers $(g = 0)$, only few weak graft *k*-mers $(g' \leq 6)$, but at least some (strong) host *k*-mers $(h \geq 6)$, in which case the read is classified as "host". A symmetric rule exists for the "graft" class, of course. An even more complex rule tests whether there is sufficient overall evidence for host but only little strong graft evidence in absolute terms, and little weak graft evidence in comparison to the host evidence. For categories "both" and "neither", a relatively large number of corresponding *k*-mers is required. Category "ambiguous" is always chosen if no "else" rule exists and no other rule applies in any given node. The thresholds have been iteratively hand-tuned on several internal human, mouse and bacterial datasets that were not part of the evaluation datasets. The thresholds are optimized for typical high-quality short reads (100–150 bp) and may have to be adjusted for long reads with higher error rates. For completeness, the Python source of the classification function appears in Table 6 in Appendix.

Quick mode

Inspired by a feature of the *kallisto* software [13] for transcript expression quantification, we additionally implemented a "quick mode" that initially looks only at the type of the third and third-last *k*-mer in every read. If the two (for single-end reads) or four (for paired-end reads)

Table 2 Properties of the *k*-mer index for different values of *k*

k-mers	$k = 23$	(%)	$k = 25$	(%)	$k = 27$	(%)
Total	4,396,323,491	(100)	4,496,607,845	(100)	4,576,953,994	(100)
Host	1,924,087,512	(43.8)	2,050,845,757	(45.6)	2,105,520,461	(46.0)
Graft	2,173,923,063	(49.4)	2,323,880,612	(51.7)	2,395,147,724	(52.3)
Both	18,701,862	(0.4)	12,579,160	(0.3)	9,627,252	(0.2)
Weak host	132,469,231	(3.0)	52,063,110	(1.2)	32,445,717	(0.7)
Weak graft	147,141,823	(3.4)	57,239,206	(1.3)	34,212,840	(0.7)

Underlying reference sequences are given in "Reference sequences" section

Table 3 Index construction

Tool	*k*	Build CPU	Build Wall	Mark CPU	Mark Wall	Total CPU	Total Wall	Mem Final	Mem Peak
Xengsort	23	50	50	591	176	641	226	12.8	17.3
Xengsort	25	53	53	437	158	490	211	15.9	20.4
Xengsort	27	51	51	495	214	546	265	17.3	21.8
Xenome	25	992	151	2338	356	3626	552	31.2	57.1
XenofilteR	–	–	–	–	–	528	658	13.0	22.0

CPU times and wall clock times in minutes and memory in Gigabytes using different tools and different *k*-mer sizes for *xengsort*. "Build" times refer to collecting and hashing the *k*-mers according to species, but without marking weak *k*-mers. "Mark" times refer to marking weak *k*-mers. "Total" times are the sum of build and mark times, plus additional I/O times. "CPU" times measure total CPU work load (as reported by the time command as user time), and "wall" times refer to actually passed time. Final size ("mem final") is measured by index size on disk (GB). Memory peak ("mem peak") is the highest memory usage during construction (GB)

types agree (e.g. all are "graft"), the fragment is classified on this sampled evidence alone. This results in quicker processing of large FASTQ files, but only considers a small sample of the available information.

Results

We evaluate our alignment-free xenograft sorting approach and its implementation *xengsort* for the common case of human-tumor-in-mouse xenografts, by using mouse datasets, human datasets, xenograft datasets and datasets from other species, and compare against an existing tool with the same purpose, *xenome* from the *gossamer* suite [7], and against a representative of alignment-based filtering tools, *XenofilteR* [2]. The hardware used for the benchmarks was one server with two AMD Epyc 7452 CPUs (with 32 cores and 64 threads each), 1024 GB DDR4-2666 memory and one 12 TB HDD with 7200 rpm and 256 MB cache.

We first report on statistics and efficiency of index construction ("Hash table construction" section), then discuss classification accuracy on several datasets ("Classification results" section), and finally compare running times ("Running times" section).

Hash table construction

Table size and uniqueness of k-mers

We evaluated $k \in \{23, 25, 27\}$ and then decided to use $k = 25$ because it offers a good compromise between species specificity and memory requirements. Table 2 shows several index properties. In particular, moving from $k = 25$ to $k = 27$, the small decrease in *k*-mers that map to both genomes and in weak *k*-mers did not justify the additional memory requirements. In addition, shorter *k*-mers lead to better error tolerance against sequencing errors, as each error affects up to *k* of the *k*-mers in a read.

Construction time and memory

Table 3 shows time and memory requirements for building the *k*-mer hash table or FM index for *bwa* (for *XenofilteR*). The main difference is that the BWA index is a succinct representation of the suffix array of the references and not a *k*-mer hash table. Our hash table construction is not paralellized; hence CPU times and wall clock times agree and are less than one hour. The hash construction of *xenome* is paralellized; we gave it 8 threads (but 9 were sometimes used); yet it does about 20 times the CPU work and takes three times as long as *xengsort*, even when using multiple threads.

Marking weak or marginal *k*-mers is paralellized in both approaches; wall clock times are measured using 8 threads. Again, *xengsort* finds the weak *k*-mers faster, both in terms of total CPU work and wall clock time.

The indexing method of bwa is not comparable, as it builds a complete suffix array (FM index) that is independent of *k* and does not include marking weak *k*-mers.

Here the CPU time is lower than the wall clock time, which indicates an I/O starved process.

We note that *xenome* uses a large amount of memory during hash table construction (it was given up to 64 GB). It works with less if restricted, but at the expense of longer running times. BWA indexing also needs significant additional memory during construction. The additional memory required by *xengsort* results from the additional sorted *k*-mer list required for detecting weak *k*-mers. Overall, our construction is fast (even though serial only) and uses a reasonable amount of memory.

Load factor and hash choice distribution

As explained in "Multi-way bucketed quotiented Cuckoo hashing" section, 3-way Cuckoo hash tables support very high loads (fill ratios) over 99.9%. However, such loads come at the expense of distributing all *k*-mers almost evenly across hash function choices. For faster lookup, it is beneficial to leave part of the hash table empty. We used a load factor of 88% and thus find 76.7% of the *k*-mers at their first bucket choice, 15.5% at their second choice and only 7.8% at their third choice, yielding an average of 1.31 lookups for a present *k*-mer.

Applying assignment optimization [11], which takes an additional 5 h (serial CPU time, not parallelized) and temporarily needs over 80 GB of RAM, we achieve a slightly better average of 1.17 lookups for a present *k*-mer.

Classification results

We applied our method *xengsort*, *xenome* and *XenofilteR* to several datasets with reads of known origin (except possible contamination issues or technical artefacts), that however present certain particular challenges. Each of the following paragraphs discusses one dataset.

Human-captured mouse exomes

A recent comparative study [1] made five *mouse exomes* accessible, which were captured with a *human-exome capture kit* and hence presents mouse reads that are biased towards high similarity with human reads. The mouse strains were A/J (two mice), BALB/c (one mouse), and C57BL6 (two mice); they were sequenced on the Illumina HiSeq 2500 platform, resulting in 11.8 to 12.7 Gbp. The datasets are available under accession numbers SRX5904321 (strain A/J, mouse 1), SRX5904320 (strain A/J, mouse 2), SRX5904319 (strain BALB/c, mouse 1), SRX5904318 (strain C57BL/6, mouse 1) and SRX5904322 (strain C57BL/6, mouse 2). Ideally, all reads should be classified as mouse reads.

Table 4 shows detailed classification results and running times. Considering the BALB/c and C57BL/6 strains first, it is evident that classification accuracy is high (over 98.9% mouse for *xengsort*, over 97.4% for *xenome*; with less than 0.64% human reads for both tools). The main difference between the tools is that *xenome* is more conservative, assigning a larger fraction of reads to the "ambiguous" (unclassified) category. With *xenome*, this happens for reads that contain two *k*-mers x, y, where x maps uniquely to human and y maps uniquely to mouse. The decision rule of *xengsort* is more permissive and tolerant towards small inconsistencies. Therefore, *xengsort* assigns more reads correctly to mouse, and fewer to the ambiguous category. Additionally, *xengsort* assigns fewer reads incorrectly to human.

However, the two samples of strain A/J give different results. Both *xengsort* and *xenome* assign a large fraction of reads (around 21% and 3.6% in the two samples) to the human genome, while *XenofilteR* assigns only 10.5% and 2.7%, respectively. While *xengsort* does assign more reads to mouse, it also assigns more reads to human, following its strategy of leaving fewer reads unassigned (ambiguous). Inspection of these reads revealed that almost all of them are low-complexity, i.e. consist of repetitive sequence, and a check with BLAT [14] revealed no hits in mouse and several gapped hits in the human genome. So the classification as human reads is not incorrect from a technical standpoint, but in fact these reads appear to point to techincal problems during then enrichment step of the library generation. An additional low-complexity filter would remove most problematic reads.

Human genome (GIAB) matepair library

We obtained FASTQ files of an Illumina-sequenced 6kb matepair library from the Genome In A Bottle (GIAB) Ashkenazim trio dataset according to the provided sequence file index[1]. The data represents a family (mother, father, son). Ideally, we see only human reads.

Figure 4a shows the classification results for *xengsort* and *xenome*. *XenofilteR* reported that the BAM files were too large to be processed and did not give a result (400 GB total for human and mouse; each BAM file over 30 GB in size). We see that almost all reads are correctly identified as human, while a small fraction is neither, which could be adapter dimers or other technical issues. However, *xenome* classifies a similarly small fraction as ambiguous. Both alignment-free tools accurately recognize that this is a pure human dataset.

Chicken genome

We obtained a paired-end (2x101bp) Illumina whole genome sequencing run of a chicken genome from a whole blood sample (accession SRX6911418) with a total of 251

[1] ftp://ftp-trace.ncbi.nlm.nih.gov/giab/ftp/data_indexes/AshkenazimTrio/sequence.index.AJtrio_Illumina_6kb_matepair_wgs_08032015.

Table 4 Detailed classification results on five human-captured mouse exomes from different mouse strains (2× A/J, 1× BALB/c, 2× C57BL/6)

A/J-1	*xengsort*		*xenome*		XfR	
Time	70 Cm	14 Wm	371 Cm	45 Wm	56 Cm	56 Wm
	Fragmets	(%)	Fragmets	(%)	Fragmets	(%)
Mouse	46,648,014	(78.03)	45,759,814	(76.54)		
Both	120,808	(0.20)	65,269	(0.11)		
Human	12,813,583	(21.43)	12,500,844	(20.91)	6,315,955	(10.56)
Ambgs.	58,449	(0.10)	1,383,547	(2.31)		
Neither	143,775	(0.24)	75,155	(0.13)		
A/J-2	*xengsort*		*xenome*		XfR	
Time	70 Cm	15 Wm	416 Cm	50 Wm	67 Cm	67 Cm
	Fragmets	(%)	Fragmets	(%)	Fragmets	(%)
Mouse	60,255,189	(95.57)	59,135,489	(93.80)		
Both	151,396	(0.24)	89,089	(0.14)		
Human	2,301,384	(3.65)	2,271,131	(3.60)	1,718,545	(2.73)
Ambgs.	57,827	(0.09)	1,340,814	(2.13)		
Neither	279,556	(0.44)	208,829	(0.33)		
BALB/c	*xengsort*		*xenome*		XfR	
Time	68 Cm	15 Wm	392 Cm	45 Wm	61 Cm	61 Wm
Mouse	62,235,960	(98.99)	61,274,277	(97.46)		
Both	118,541	(0.19)	68,949	(0.11)		
Human	342,908	(0.55)	348,154	(0.55)	285,556	(0.45)
Ambgs.	45,063	(0.07)	1,098,036	(1.65)		
Neither	127,035	(0.20)	80,091	(0.13)		
C57BL/6-1	*xengsort*		*xenome*		XfR	
Time	72 Wm	14 Wm	359 Wm	44 Wm	58 Cm	58 Wm
Mouse	57,993,361	(98.93)	57,522,446	(98.13)		
Both	118,984	(0.20)	74,325	(0.13)		
Human	375,716	(0.64)	376,653	(0.64)	290,894	(0.50)
Ambgs.	27,731	(0.05)	571,542	(0.98)		
Neither	103,895	(0.18)	74,721	(0.13)		
C57BL/6-2	*xengsort*		*xenome*		XfR	
Time	67 Cm	15 Wm	422 Cm	51 Wm	62 Cm	62 Wm
Mouse	62,384,448	(99.00)	61,941,783	(98.30)		
Both	107,019	(0.17)	66,163	(0.10)		
Human	189,536	(0.30)	208,149	(0.33)	132,535	(0.21)
Ambgs.	27,142	(0.04)	562,659	(0.89)		
Neither	304,677	(0.48)	234,068	(0.37)		

Running times are reported both in CPU minutes (Cm), measuring CPU work, and wall clock minutes (Wm), measuring actual time spent. Times for *XenofilteR* (XfR) do not include alignment or BAM sorting time. Classification results report the number and percentage (in brackets) of fragments classified as mouse (correct), both human and mouse (likely correct), human (incorrect), ambiguous (no statement) and neither (likely incorrect). *XenofilteR* (XfR) only extracts human fragments and does not classify the remainder; so only the number of fragments classified as human are reported

million paired-end reads. Ideally, none of these reads are recognized as mouse or human reads. Figure 4b shows divergent results. For *XenofilteR*, we can only say that almost no reads are extracted as human; the remainder is unclassified. *Xenome* assigns a small number of reads to each category and only around 90% into the "neither"

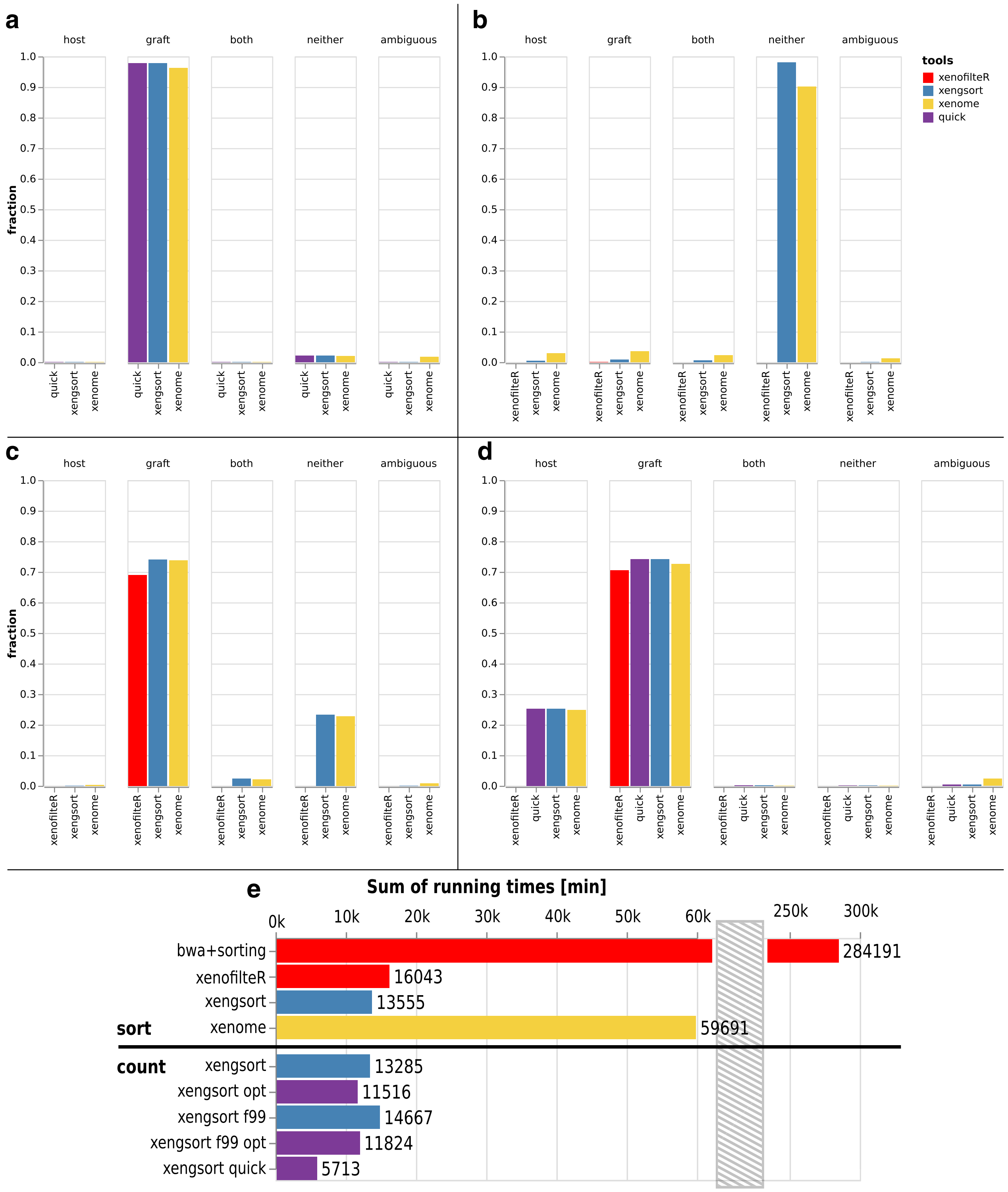

Fig. 4 Classification results of different tools (*XenofilteR*, *xenome*, *xengsort*, and partially *xengsort* with "quick" option) on several datasets: **a** GIAB human matepair dataset (*XenofilteR* did not run on this dataset); **b** Chicken genome; **c** Human lymphocytic leukemia RNA-seq data; **d** Patient-derived xenograft (PDX) RNA-seq data. **e** CPU times on the PDX RNA-seq dataset with different tools and different *xengsort* parameters (see text)

Table 5 Dataset sizes (number of fragments; M: millions) and CPU times in minutes spent on different datasets, measured with the "time" command (user time) when running with 8 threads [*xenome*, *xengsort*, *bwa-mem*, BAM sorting, except for *XenofilteR* (XfR), which is single-threaded]

Dataset/tool	Size	XfR+	*bwa+*	Sort	*Xenome*	*Xengsort*
Mouse exomes	307 M	310+	8291+	179	1823	368
Human genome	1258 M	N/A+	222939+	940	9845	2463
Chicken genome	251 M	76+	6976+	118	1273	592
Leukemia RNA	1760 M	778+	22,111+	521	5188	1680
PDX RNA	9742 M	16,043+	2,78,329+	5862	59,692	13,555

N/A: not applicable; tool could not be run on this dataset

category, while *xengsort* assigns 98.11% of the reads as "neither".

Human lymphocytic leukemia tumor RNA-seq data

We obtained single-end FASTQ files from RNA-seq data of 5 human T-cell large granular lymphocytic leukemia samples, where recurrent alterations of TNFAIP3 were observed, and 5 matched controls (13.4 Gbp to 27.5 Gbp). The files are available from SRA accession SRP059322 (datasets SRX1055051 to SRX1055060). Surprisingly, not all fragments were recognized as originating from human tissue (Fig. 4c). While *xenome* and *xengsort* agreed that the human fraction is close to 75%, *XenofilteR* assigned fewer reads to human origins (less than 70%).

For this and the other RNA-seq datasets, we trimmed the Illumina adapters using cutadapt [15] prior to classification, as some RNA fragments may be shorter than the read length. If this step is omitted, even fewer fragments are classified as human (graft): just below 70% for *xenome* and *xengsort*, and only about 53% for *XenofilteR*. The number of fragments classified as neither increases correspondingly.

We investigated the reads classified by *xengsort* as neither human nor mouse. Quality control with FastQC [16] revealed nothing of concern, but showed an unusual bimodal per-fragment GC content distribution with peaks at 45% and 55%. BLASTing the fragments against the non-redundant nucleotide database [17] yielded no hits at all for 97% of these fragments. A small number (2%) originated from the bacteriophage PhiX, which was to be expected, because it is a typical spike-in for Illumina libraries. The remaining 1% of fragments showed random hits over many species without a distinctive pattern. We therefore concluded that the "neither; ; fragments mainly consisted of artefacts from library construction, such as ligated and then sequenced random primers.

Patient-derived xenograft (PDX) RNA-seq samples from human pancreatic tumors

We evaluated 174 pancreatic tumor patient-derived xenograft (PDX) RNA-seq samples that are available internally at University Hospital Essen. Figure 4d shows that all three tools classify between 70% and 74% as graft (human) fragments. Again, *XenofilteR* seems to be the most conservative tool with about 70%, and *xenome* classifies about 72% as human and *xengsort* 74%. The remaining reads are not classified by *XenofilteR*, while *xenome* and *xengsort* both assign about 25% to host (mouse). Furthermore, *xenome* classifies about 2% and *xengsort* less than 1% as ambiguous.

So we observe that on all datasets, *xengsort* is more decisive than *xenome* and, judging from the pure human and mouse datasets, mostly correct about it. Because this is a large dataset, we also applied *xengsort*'s quick mode and found essentially no differences in classification results (less than 0.001 percentage points in each class; e.g. for graft: quick 74.0111% vs. standard 74.0105% of all reads; difference 0.0006%; cf. Fig. 4d).

Running times

A summary of running times for all datasets appears in Table 5.

Human-captured mouse exomes

Our implementation *xengsort* needs around 70 CPU minutes for each of the five human-captured mouse exomes (total: 368 min), and less than 15 min of wall clock time using 8 threads. The speed-up being less than 8 results from serial intermediate I/O steps. While *xenome* makes better use of parallelism, it is slower overall, requiring 5 to 6 times the CPU work of *xengsort*. For only scanning already aligned BAM files, *XenofilteR* is surprisingly slow, and we see that we can sort the reads from scratch in almost the same amount of CPU work that is required to compare (already computed) alignment scores. When adding bwa mem alignment times (even without the time required for sorting the resulting BAM files), *XenofilteR* needs an additional 887 to 1424 CPU minutes for the human alignments and an additional 424 to 777 min for the mouse alignments per dataset, making

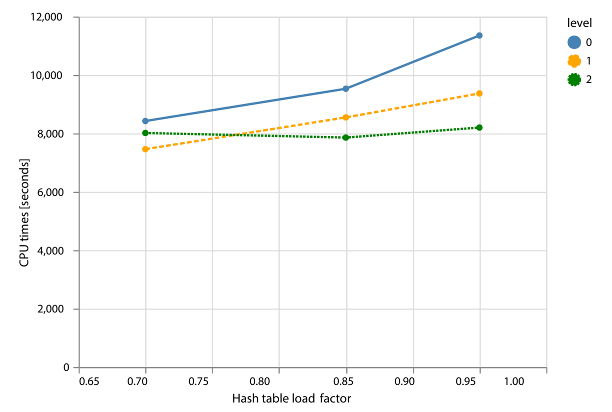

Fig. 5 Effect of different prefetching levels on the running time of the adversarial chicken genome dataset: no prefetching (0, default), prefetching the second bucket (1), or prefetching both second and third bucket (2). Times are averages over 4 repeated runs

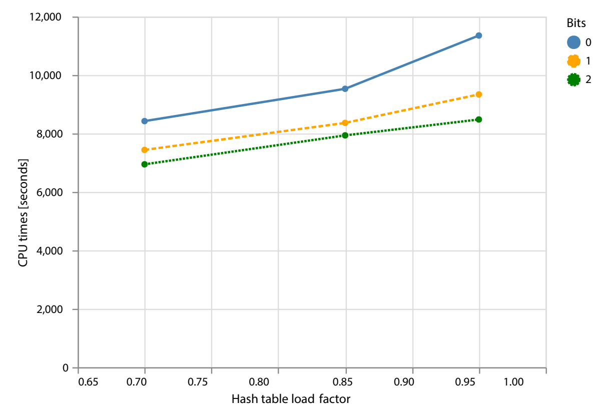

Fig. 6 Effect of shortcut bits for unsuccessful k-mer searches on the running time of the adversarial chicken genome dataset: no extra bits (0, default), one extra bit (1) or two extra bits (2); see "Performance engineering" section. Times are averages over 6 repeated runs

the alignment-based approach far less efficient than the alignment-free approach.

Human genome (GIAB) matepair library

We observe the same wall clock time ratio (about 3.5) between *xenome* and *xengsort* as for the mouse exome dataset.

Because this is a very large dataset (112 GB gzipped FASTQ), we additionally evaluated the effects of using *xengsort*'s "quick mode". We observed a significant reduction in processing time (by about 33%) and almost unchanged classification results. We also ran the *xengsort* classification with the optimized hash table (using an optimized assignment computed using the methods from [11] and found a small reduction (9%) in running time.

Chicken genome

The BAM file scan of *XenofilteR* here beats the alignment-free tools (cf. Table 5) because both BAM files are essentially empty, as very few reads align against human or mouse. Also, the speed advantage of *xengsort* over *xenome* is less on this dataset, mainly because most k-mers are not found in the index and require $h = 3$ memory lookups and likely cache misses. Such a dataset that contains neither graft nor host reads is adversarial for our design of *xengsort*. However, the engineering methods introduced in "Performance engineering" section are effective on such a dataset. The following evaluations are based on one lane (1/3) of the complete chicken dataset because of time constraints.

Figure 5 shows the effect of using different amounts of prefetching: none (0, default), prefetching the second choice bucket (1), or the second and third choice buckets (2). At low table loads (0.7), prefetching is not very helpful (level 1) or even detrimental (level 2 compared to level 1) because of the additional overhead. At intermediate load levels (0.85), prefetching helps, but a second bit does not provide an additional advantage. At high table loads (0.95), more aggressive prefetching provides an additional gain in running time. In fact, with prefetching level 2, the running time is almost independent of the load factor.

Figure 6 shows the effect of using 0 (default), 1 or 2 shortcut bits per bucket. Almost independently of the load factor, using one shortcut bit yields a measurable running time reduction by 10%. Using a second bit gives only a small additional advantage (ca. 4%).

Unfortunately, the effects of both optimizations are not cumulative. Essentially, an effective use of shortcuts renders prefetching almost useless. On the other datasets, where most k-mer queries are successful, the effects of both optimizations are much less pronounced and even negligible.

Human lymphocytic leukemia tumor RNA-seq data

Again, *xengsort* is more than 3 times faster than *xenome* and needs time comparable to *XenofilteR* even when only the time for sorting and scanning the existing BAM files is taken into account (Table 5). Producing the alignments for *XenofilteR* takes much longer.

Patient-derived xenograft (PDX) RNA-seq samples from human pancreatic tumors

With its 174 samples, this is a particularly large dataset of the type that we optimized *xengsort* for. Therefore, running time differences between the three methods become particularly apparent. Figure 4e shows that the alignment using *bwa-mem* and the sorting of the BAM file for *XenofilteR* took over 284,191 CPU minutes (close to 200 CPU days). After that, *XenofilteR* required an additional 16,043 CPU minutes (over 11 CPU days) to classify the aligned and sorted reads. In comparison, *xenome* with 59,691 CPU minutes (41.5 days) took only 20% of the time used by *bwa-mem* and *XenofilteR*, and *xengsort* needed 13,555 CPU minutes (9.5 CPU days) to sort all reads and is therefore even faster than the classification by *XenofilteR* alone, even excluding the alignment and sorting steps, and over 4 times faster than *xenome*. Using the "quick mode" with an optimized hash table at 88% load needed only 5713 CPU minutes (less than 4 CPU days), i.e., less than half of the time of a full analysis.

We additionally examined some trade-offs for this dataset. First, we note that only counting proportions without output ("count" operation) is not much faster than sorting the reads into different output files ("sort" operation): 13,285 vs. 13,555 CPU minutes (2% faster). We additionally measured the running time of *xengsort*'s count operation on hash tables with different load factors (88% and 99%) using both the standard assignment by random walk and an optimal assignment [11]. As expected, a load factor of 99% was slower than 88% (by 10.4% on the random walk assignment, but only by 2.6% on the optimized assignment). Using the optimal assignment gives a speed boost (13.3% faster at 88% load; 19.3% at 99% load). The optimized assignment at 99% load yields an even faster running time than the random walk assignment at 88% load by 11% (11,824 vs. 13,285 CPU minutes).

Discussion and conclusion

We revisited the xenograft sorting problem and improved upon the state of the art in alignment-free methods with our implementation of *xengsort*.

On typical datasets (PDX RNA-seq), it is at least four times faster and needs less memory than the comparable *xenome* tool. Our experiments show that *xengsort* provides accurate classification results, and classifies more reads than *xenome*, which more often assigns the label "ambiguous". Surprisingly, on PDX datasets, our approach is even faster than scanning already aligned BAM files. This favorable behavior arises because almost every *k*-mer in every read can be expected to be found in the key-value store, and lookups of present keys are highly optimized.

On adversarial datasets (e.g., a sequenced chicken genome, where almost none of the *k*-mers can be found in the hash table), *xengsort* is twice as fast as *xenome*, but 8 times slower than scanning pre-aligned and pre-sorted BAM files (which are mostly empty). With additional engineering tweaks, such as shortcut bits or software prefetching, our performance on such datasets can be improved (10% speed gain). More refined prefetching strategies, such as *k*-mer look-ahead, may lead to further improvements, and we will experiment with additional ideas.

Given that producing and sorting the BAM files takes significant additional time, our results show that overall, alignment-free methods require significantly less computational resources than alignment-based methods. In view of the current worldwide discussions on climate change and energy efficiency, we advocate that the most resource-efficient available methods should be used for a task, and we propose that *xengsort* is preferable to existing work in this regard. Even though one could argue that alignments are needed later anyway, we find that this is not always true: First, to analyze PDX samples, typically only the graft reads are further considered and need to be aligned. Second, recent research has shown that more and more application areas can be addressed by alignment-free methods, even structural variation and variant calling [18], so alignments may not be needed at all.

On the methodological side, we developed a general key-value store for DNA/RNA *k*-mers that allows extremely fast lookups, often only a single random memory access, and that has a low memory footprint thanks to a high load factor and the technique of quotienting.

Thus this work might be seen as a blueprint for implementations of other alignment-free methods, such as for gene expression quantification, metagenomics, etc. In principle, one could replace the underlying key-value store of each published *k*-mer based method by the hashing approach presented here and probably obtain a speed-up of factor 2 to 4, while at the same time saving some space for the hash table. In practice, such an approach may be difficult because the code in question is often deeply nested in the application. However, we would like to suggest that for future implementations, three-way bucketed Cuckoo hash tables with quotienting should be given serious consideration.

A (small) limitation of our approach is that the size of the hash table must be known (at least approximately) in advance. In principle we could grow the table dynamically, but it means re-hashing all elements. Fortunately, the total length of the sequences in the *k*-mer key-value store provides an easily calculated upper bound. The advantage of such a static approach is that only little additional memory is required during construction.

The software *xengsort* is available at http://gitlab.com/genomeinformatics/xengsort under the MIT license. Installation and usage instructions are provided within the README file of the repository. The software is written in Python, but makes use of just-in-time compilation using the numba package [19]. While requiring an additional 1–2 s of startup time, this allows for many optimizations, because certain parameters that become

Table 6 Python source code of xengsort's classification routine with thresholds, as of v1.0.0

```
def classify_xengsort(counts):
    # counts=[neither,host,graft,both,0,weakhost weakgraft,both]
    # returns: 0=host, 1=graft, 2=ambiguous, 3=both, 4=neither
    nkmers = 0
    for i in counts:
        nkmers += i
    if nkmers == 0:
        return 2  # no k-mers -> ambiguous
    nothing = uint32(0)
    few = uint32(6)
    insubstantial = uint32(nkmers // 20)
    Ag = uint32(3)
    Ah = uint32(3)
    Mh = uint32(nkmers // 4)
    Mg = uint32(nkmers // 4)
    Mb = uint32(nkmers // 5)
    Mn = uint32((nkmers * 3) // 4 + 1)

    hscore = counts[1] + counts[5] // 2
    gscore = counts[2] + counts[6] // 2

    # no host
    if counts[1] + counts[5] == nothing: # no host
        if gscore >= Ag:
            return 1  # graft
        if counts[3] + counts[7] >= Mb: # both
            return 3  # both
        if counts[0] >= Mn: # neither
            return 4  # neither

    # host, but no graft
    elif counts[2] + counts[6] == nothing: # no graft
        if hscore >= Ah:
            return 0  # host
        if counts[3] + counts[7] >= Mb: # both
            return 3  # both
        if counts[0] >= Mn: # neither
            return 4  # neither

    # some real graft, few weak host, no real host:
    if counts[2] >= few and counts[5] <= few and counts[1] == nothing:
        return 1  # graft
    # some real host, few weak graft, no real graft:
    if counts[1] >= few and counts[6] <= few and counts[2] == nothing:
        return 0  # host

    # substantial graft, insubstantial real host,
    # a little weak host compared to graft:
    if (counts[2] + counts[6] >= Mg and counts[1] <= insubstantial and
                                        counts[5] < gscore):
        return 1  # graft
    # substantial host, insubstantial real graft,
    # a little weak graft compared to host:
    if (counts[1] + counts[5] >= Mh and counts[2] <= insubstantial and
                                        counts[6] < hscore):
        return 0  # host
    # substantial both, insubstantial host and graft:
    if (counts[3] + counts[7] >= Mb and gscore <= insubstantial and
                                       hscore <= insubstantial):
        return 3  # both
    # substantial neither:
    if counts[0] >= Mn:
        return 4  # neither
    # no specific rule applies:
    return 2  # ambiguous
```

only known at run time, such as random parameters for the hash functions, can be compiled as constants into the code. These optimizations yield savings that exceed the initial compilation effort.

While we have indications that classification results agree well overall among all methods and variants, we concur with a recent study [1] that there exist subtle differences, whose effects can propagate through computational pipelines and influence, for example, variant calling results downstream, and we believe that further evaluation studies are necessary. In contrast to their study, we however suggest that a best practice workflow for PDX analysis should start (after quality control and adapter trimming on RNA-seq data) with *alignment-free* xenograft sorting, followed by aligning the graft reads and the reads that can originate from both genomes to the graft genome. In any workflow, the latter reads, classified as "both" may pose problems, because one may not be able to decide on the species of origin. Indeed, ultra-conserved regions of DNA sequence exist between human and mouse. In this sense we believe that full read sorting (into categories host, graft, both, neither, ambiguous, as opposed to extracting graft reads only) gives the highest flexibility for downstream steps and is preferable to filter-only approaches.

Appendix

Table 6 shows the Python source of the read (pair) classification routine. The input vector `counts` corresponds to $(x, h, g, b_1, 0, h', g', b_2)$ with $b = b_1 + b_2$ in the notation of "Fragment classification" section.

Acknowledgements
We thank Uriel Elias Wiebelitz for preliminary experiments of the effectiveness of prefetching and Elias Kuthe for helping with the prefetching implementation.

Authors' contributions
SR provided the initial concept. JZ designed and implemented the software and performed the evlauations. Both authors wrote and edited the manuscript. Both authors read and approved the final manuscript.

Funding
Open Access funding enabled and organized by Projekt DEAL. S.R. is grateful for funding from DFG SFB 876 subproject C1, and Mercator Research Center Ruhr (MERCUR), project Pe-2013-0012.

Availability of data and materials
All datasets mentioned in this article are publicly available from third parties; their accession numbers are given in the respective paragraph.

Declarations

Competing interests
The authors declare that they have no competing interests.

Author details
[1] Bioinformatics, Computer Science XI, TU Dortmund University, Dortmund, Germany. [2] Genome Informatics, Institute of Human Genetics, University Hospital Essen, University of Duisburg-Essen, Essen, Germany.

Received: 2 February 2021 Accepted: 24 March 2021
Published online: 02 April 2021

References
1. Jo SY, Kim E, Kim S. Impact of mouse contamination in genomic profiling of patient-derived models and best practice for robust analysis. Genome Biol. 2019;20(1):231.
2. Kluin RJC, Kemper K, Kuilman T, de Ruiter JR, Iyer V, Forment JV, Cornelissen-Steijger P, de Rink I, Ter Brugge P, Song JY, Klarenbeek S, McDermott U, Jonkers J, Velds A, Adams DJ, Peeper DS, Krijgsman O. XenofilteR: computational deconvolution of mouse and human reads in tumor xenograft sequence data. BMC Bioinform. 2018;19(1):366.
3. Giner G. XenoSplit. Unpublished; 2019. source code available at https://github.com/goknurginer/XenoSplit.
4. Khandelwal G, Girotti MR, Smowton C, Taylor S, Wirth C, Dynowski M, Frese KK, Brady G, Dive C, Marais R, Miller C. Next-generation sequencing analysis and algorithms for PDX and CDX models. Mol Cancer Res. 2017;15(8):1012–6.
5. Ahdesmäki MJ, Gray SR, Johnson JH, Lai Z. Disambiguate: an open-source application for disambiguating two species in next generation sequencing data from grafted samples. F1000Res. 2016;5:2741.
6. Bushnell B. BBsplit, Joint Genome Institute, Walnut Creek, CA. Part of BBTools; 2014–2020. https://jgi.doe.gov/data-and-tools/bbtools/.
7. Conway T, Wazny J, Bromage A, Tymms M, Sooraj D, Williams ED, Beresford-Smith B. Xenome—a tool for classifying reads from xenograft samples. Bioinformatics. 2012;28(12):172–8.
8. Callari M, Batra AS, Batra RN, Sammut SJ, Greenwood W, Clifford H, Hercus C, Chin SF, Bruna A, Rueda OM, Caldas C. Computational approach to discriminate human and mouse sequences in patient-derived tumour xenografts. BMC Genomics. 2018;19(1):19.
9. Dai W, Liu J, Li Q, Liu W, Li YX, Li YY. A comparison of next-generation sequencing analysis methods for cancer xenograft samples. J Genet Genomics. 2018;45(7):345–50.
10. Walzer S. Load thresholds for Cuckoo hashing with overlapping blocks. In: Chatzigiannakis I, Kaklamanis C, Marx D, Sannella D, editors. 45th international colloquium on automata, languages, and programming, ICALP 2018. LIPIcs; 2018. vol. 107, p. 102–110210. Schloss Dagstuhl-Leibniz-Zentrum fuer Informatik, Wadern, Germany. https://doi.org/10.4230/LIPIcs.ICALP.2018.102
11. Zentgraf J, Timm H, Rahmann S. Cost-optimal assignment of elements in genome-scale multi-way bucketed Cuckoo hash tables. In: Proceedings of the symposium on algorithm engineering and experiments (ALENEX) 2020, 2020, p. 186–98. SIAM, Philadelphia, PA, USA. https://doi.org/10.1137/1.9781611976007.15
12. Espinosa A. Cuckoo breeding ground—a better cuckoo hash table; 2018. https://cbg.netlify.app/publication/research_cuckoo_cbg/.
13. Bray NL, Pimentel H, Melsted P, Pachter L. Near-optimal probabilistic RNA-seq quantification. Nat. Biotechnol. 2016;34(5): 525–7. Erratum in Nat. Biotechnol. 2016;34(8):888.
14. Kent WJ. BLAT—the BLAST-like alignment tool. Genome Res. 2002;12(4):656–64.
15. Martin M. Cutadapt removes adapter sequences from high-throughput sequencing reads. EMBnet J. 2011;17(1):10–2. https://doi.org/10.14806/ej.17.1.200.
16. Andrews S. FastQC: a quality control tool for high throughput sequence data. Babraham Bioinformatics, Inc; 2010. http://www.bioinformatics.babraham.ac.uk/projects/fastqc/.
17. Camacho C, Coulouris G, Avagyan V, Ma N, Papadopoulos J, Bealer K, Madden TL. BLAST+: architecture and applications. BMC Bioinform. 2009;10:421.
18. Standage D.S, Brown C.T, Hormozdiari F. Kevlar: a mapping-free framework for accurate discovery of de novo variants. iScience. 2019;18:28–36.
19. Lam SK, Pitrou A, Seibert S. Numba: a LLVM-based python JIT compiler. In: Finkel H, editor. Proceedings of the second workshop on the LLVM compiler infrastructure in HPC, LLVM 2015; 2015, p. 7–176. New York: ACM. https://doi.org/10.1145/2833157.2833162.