Heterogeneous Seeds: Improving the Speed, Efficiency and Error Tolerance of Seed-and-Extend Type Read Mappers

Hongyi Xin[1], Hitesh Arora[1,4], Farhad Hormozdiari[2], Can Alkan[3*], and Onur Mutlu[1*]

[1]Dept. of Electrical and Computer Engineering, Carnegie Mellon University, Pittsburgh, PA, 15213, United States. [2]Dept. of Computer Science, University of California Los Angeles, Los Angeles, CA, 90095, United States. [3]Dept. of Computer Engineering, Bilkent University, Ankara, 06800, Turkey. [4]Dept. of Computer Science and Engineering, Indian Institute of Technology Guwahati, Guwahati, AS, 781039, India.

Email: Can Alkan*- calkan@cs.bilkent.edu.tr; Onur Mutlu*- onur@cmu.edu;

*Corresponding author

Abstract

With advances in high throughput sequencing technologies (HTS), billions of short DNA fragments (also known as "reads") are processed *in silico* for genome analysis at a higher rate. To keep up with the high throughput of the sequencing platforms and to tolerate more errors with shorter fragments, a faster and error-tolerant DNA read mapper is required.

Many popular modern mappers require the reference genome to be indexed with different algorithms and data structures in order to enable fast mapping and high error tolerance. Unfortunately, these mappers are either fast, error tolerant or memory-efficient but not all three at once. We find that this is fundamentally due to the different seed lengths the mappers employ, which is the unit length (in terms of the number of bases) that a mapper uses to index the reference genome. Mappers that use short seeds (10$\sim$14 bases) are usually more tolerant to errors, and more memory efficient but also slow whereas mappers that use long seeds (over 20 bases) are usually fast, but either less tolerant to errors or less memory efficient. Our goal in this paper is to design an algorithm and a data structure that obtain the benefits of both short and long seeds, i.e., fast, tolerant to errors and memory-efficient.

In this paper, we provide a detailed analysis of the effects of using different seed lengths on the speed, error tolerance and memory efficiency of conventional mappers and conclude that we need to use both short and long seeds to obtain the benefits of both. Based on the analysis, we propose simultaneously using seeds with

different lengths in a mapper. We call this idea "Heterogeneous Seeds". We also propose a data structure, called the"Heterogeneous Lookup Table", and a fragment dividing algorithm, called "Jigsaw Seeds and Overlapping Seeds" to enable mapping with "Heterogeneous Seeds".

Our experimental evaluations show that "Heterogeneous Seeds" reduce computation overhead up to 14.8-fold compared to short seeds at 12 bases (2x compared to long seeds at 20 bases); retain the same level of error tolerance as short seeds (provide 2x higher error-tolerance compared to long seeds), while increasing the memory overhead by only 1.5% compared to short seeds (they reduce memory overhead by at least 2x compared to long seeds). We conclude that "Heterogeneous Seeds" approximate the benefits of both short seeds and long seeds.

Introduction

In the past decade, the emergence of new sequencing technologies triggered a revolution in the field of genomics. These massively parallel sequencing technologies, commonly known as high-throughput sequencing (HTS) platforms, are able to sequence a mammalian size genome in a few days, enabling applications such as investigating human genome diversity between populations [1], finding genomic variants that are likely to cause diseases [2–7], and learning the genomes of the great ape species [8–13] and even ancient hominids [14, 15] to understand human evolution. Despite the advantages these new sequencing platforms offer, they increase the computational burden of sequence mapping, which is one of the main post processing procedures in reconstructing the genome from raw outputs produced from the sequencing platforms. Specifically, HTS platforms impose three computational challenges: 1) the vast quantity of sequenced data subjected to mapping, which increases the workload of mapping, 2) shorter read lengths and less information per unit piece of data, which increase the difficulty of mapping and 3) higher sequencing errors when compared to the traditional capillary-based sequencing, which decreases the precision of mapping.

We categorize the current state-of-the-art mappers into two categories: *suffix-array based mappers* and *hash-based mappers*. Suffix-array mappers (based on BWT-FM [16, 17]), provide superior mapping speed faster, but they typically suffer greatly from increased number of errors. On the other hand, hash-based mappers are very resilient to errors and are capable of searching for all possible mappings of a read within a certain number of errors [18] but they are also very slow because short seeds tend to appear at multiple locations in the reference genome. Not surprisingly, the efficiency and power of hash-based mappers directly depend on the length of the selected seeds.

In this paper, we provide a detailed analysis of how different seed lengths affect the speed, the memory

usage and the error-tolerance of a mapper. We propose a new metric, "mapping cost", to estimate the amount of computation different seed lengths lead to. We observe that the few expensive short seeds, i.e. seeds that are very frequent in the reference genome, are responsible for the large mapping cost of short seeds. We then demonstrate that by extending only the expensive short seeds into cheap longer seeds which become cheap at different lengths, the mapper achieves the maximum reduction in mapping cost at the minimum increase in memory overhead. Hence, we conclude that it is optimal to apply different lengths to different seeds according to their frequencies in the reference genome in order to achieve both high mapping speed, low memory overhead and high error-tolerance at the same time. We call this concept "Heterogeneous Seeds". We also propose a new data structure "Heterogeneous Lookup Table" and a novel seed dividing algorithm "Jigsaw Seeds and Overlapping Seeds" to implement the "Heterogeneous Seeds" concept.

Our experimental evaluations show that using heterogeneous seeds, we can approximate the benefits of both short seeds and long seeds: 1) low mapping cost (similar to long seeds), 2) high error-tolerance (similar to short seeds) and 3) low storage cost (similar to short seeds).

The seed length dilemma: long seeds or short seeds

An important parameter for hash-based mappers is the seed length. Most hash-based mappers use seeds at a length between 10 to 14 bases, with some exceptions which use longer seeds up to 20 bases causing higher memory usage. In this section, we propose the metric "mapping cost" that depends on the cheap vs. expensive status of seeds (Figure 1) to estimate the amount of computation different seed lengths lead to and analyze the effects of seed lengths on the speed, memory-efficiency and error tolerance of the mapper.

Mapping cost and the effects on the mapping speed of the mapper

Seed extension, or verification, through alignment is the major calculation in hash-based mappers which occupies more than 90% of the execution time through our profiling experiments. The shorter the seed is, the more frequently will it appear in the reference genome thus the more candidate locations it will return. As previous works point out [18], most of the verification computation are unnecessary since most locations of the short seeds are false locations which do not provide correct mappings of the entire long read.

In addition, the number of locations of the seeds are extremely unbalanced: while most of the seeds have fewer than a hundred locations, some seeds may be seen in thousands of locations. We call such seeds as **expensive seeds** (solid shaded entries in Figure 1).

In order to illustrate how expensive seeds hurt the performance of the mapper and to evaluate how

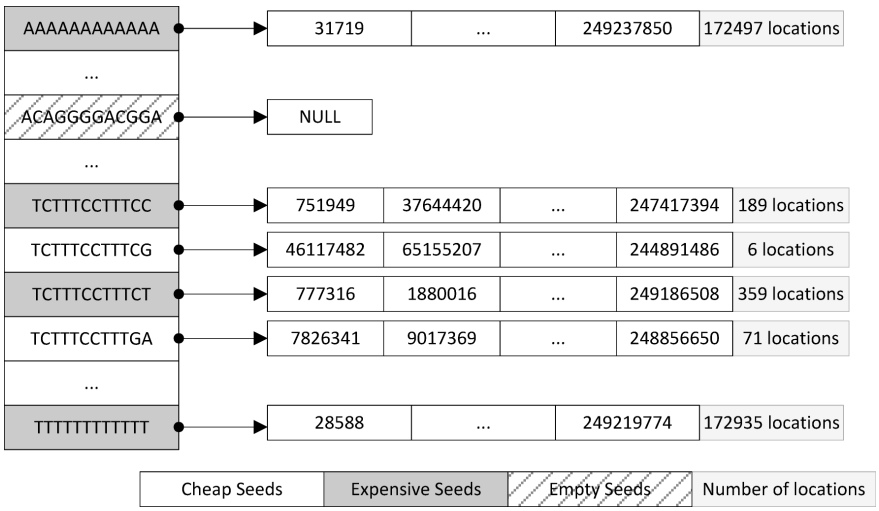


Figure 1: A typical lookup table used in hash-based mappers, which is a permutation array. Logically, the keys of the table are seeds and the values of the table are locations. In this table, most seeds have fewer than 100 locations, which are defined as "cheap seeds" while a few seeds have more than 100 locations which are defined as "expensive seeds". In reality, this is implemented as a permutation array, stores a pointer to the location list of all permutations. Permutations that have no location are defined as "empty seeds" which store a NULL pointer.

longer seeds may alleviate this problem, we propose a new metric called the **mapping cost**. The mapping cost is defined as the average number of locations of a seed selected randomly from the genome. In other words, the mapping cost is the estimation of the number of locations $E[Loc(S)]$ of a random seed S, which is calculated as the sum of products of the number of locations of a seed S_i times the probability of selecting this seed S_i through a random draw in the reference genome, as shown in Equation 1:

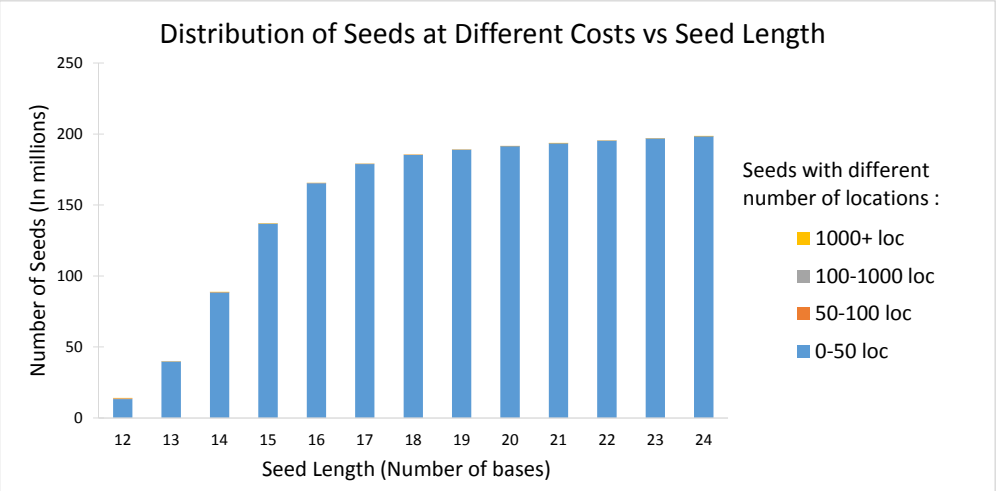
$$E[Loc(S)] = \sum_{i}^{N} P(S_i) \times Loc(S_i) \tag{1}$$

The probability of picking the seed S_i from a random locus in the genome is calculated as the number of locations of the seed S_i over the total number of locations of the entire genome, which is the length of the genome defined as a constant C: $\sum_i^N Loc(S_i) = C$. As a result, we can rewrite Equation 1 as Equation 2:

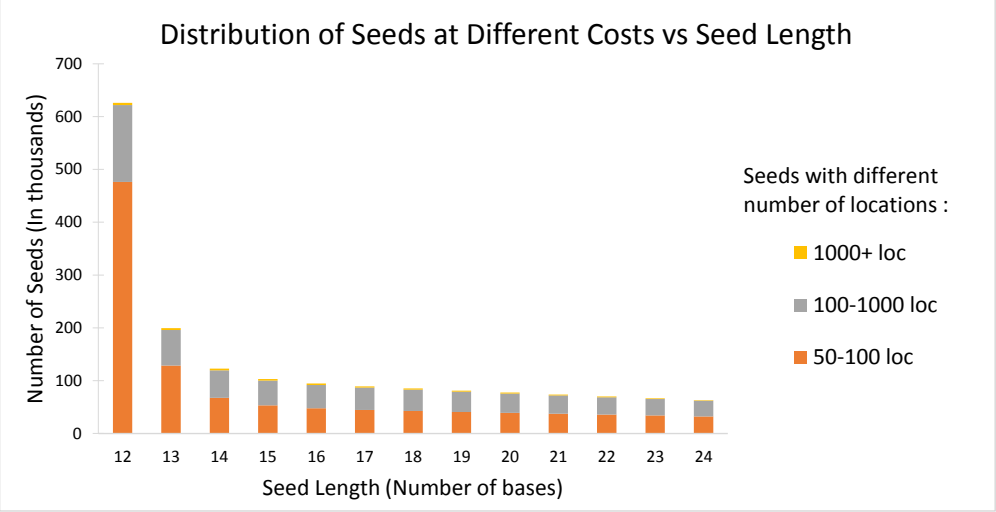
$$E[Loc(S)] = \sum_{i}^{N} P(S_i) \times Loc(S_i) = \sum_{i}^{N} \frac{Loc(S_i)}{C} \times Loc(S_i) = \frac{1}{C} \times \sum_{i}^{N} Loc(S_i)^2 \tag{2}$$

Our goal is to minimize $E[Loc(S)]$ for a genome of length C. We can formulate our objective function as a classical mathematical problem known as the Inequality of Arithmetic Mean and Root-Mean Square (a direct consequence of the Cauchy-Schwarz Inequality). The inequality says $\sqrt{\frac{x_1^2+x_2^2+\cdots+x_N^2}{N}} \geq \frac{x_1+x_2+\cdots+x_N}{N}$ with equality if and only if $x_1 = x_2 = \cdots = x_N$. In this case, if all seeds have the same number of locations, in

other words if the locations are distributed evenly into all seeds, we can reach the minimum of the estimate $E[Loc(S)] = \frac{1}{C} \times N \times (\frac{C}{N})^2 = \frac{C}{N}$, which also decreases with a larger N. This implies that we prefer a more balanced location distribution among all seeds and a larger number of seeds.



(a)



(b)

Figure 2: Figure (a) shows the distribution of seeds in different number of locations with different lengths of the human chromosome 1. Figure (b) shows the same distribution after taking out the seeds having 0-50 locations to show the reduction of expensive seeds.

A more balanced location distribution can be accomplished through increasing the seed length. When the seeds are longer, not only are there more seeds in total, which is a larger N, but also the locations are more evenly distributed among the seeds—all seeds will all together have fewer locations, generating fewer

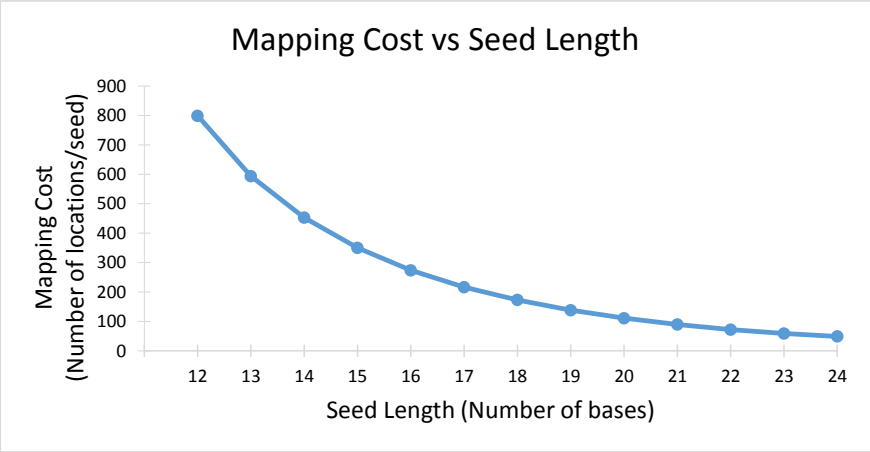


Figure 3: Mapping cost with different seed lengths of the human chromosome 1.

verification calculations during mapping, as illustrated in Figure 2. As a result, as the seed length increases, the mapping cost decreases as shown in Figure 3, which improves the mapping speed.

The effects on memory-efficiency of the mapper

As Figure 2 also shows, as the seed length increases, the *potential* number of seeds also increases. In fact, the potential number of seeds at 20 bases is 13.7x of the number of seeds at 12 bases. As a result, longer seeds require more memory space to store more data in the lookup table,

Theoretically, the size of the lookup table of length 20 should be at most 13.7x of the size of length 12 since the number of seeds of length 20 is 13.7x of the number of seeds of length 12. However, to perform hash table lookups in O(1) time, most mappers tend to index all possible permutations of seeds (Figure 1), which exponentially increases the memory requirements. Among all permutations, many of them never appeared in the reference genome hence have no locations. In the remainder of the paper, we call the seeds with zero locations as **"empty seeds"**.

Empty seeds reduce the memory-efficiency of the mapper since they occupy empty space in the array. As the seed length increases, a larger portion of the permutations becomes empty seeds, indicating reduced memory-efficiency of the mapper, as shown inFigure 4.

The effects on error-tolerance of the mapper

According to the Pigeonhole Principle, e errors can destroy at most e non-overlapping seeds, therefore we can guarantee to find an intact seed if the read is divided into more than $e+1$ non-overlapping seeds. When

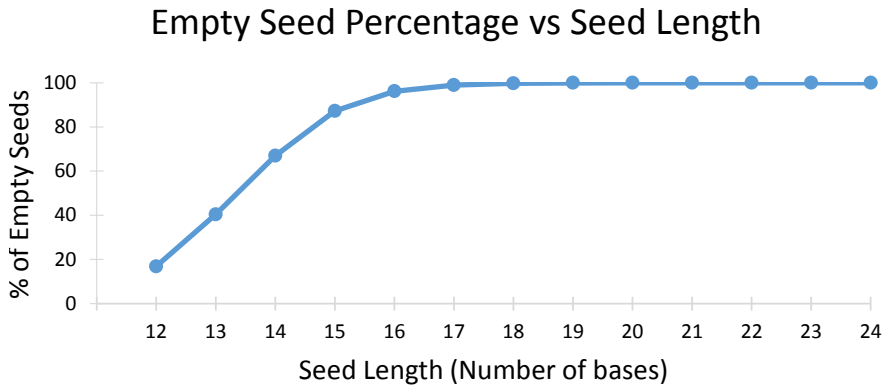


Figure 4: Percentage of empty seeds in the permutation arrays with different seed lengths of the human chromosome 1.

the seeds are short, a read can be divided into many non-overlapping seeds, which yields high error-tolerance. As the seed length increases, the number of seeds to which a read can be partitioned into decreases, reducing the error-tolerance.

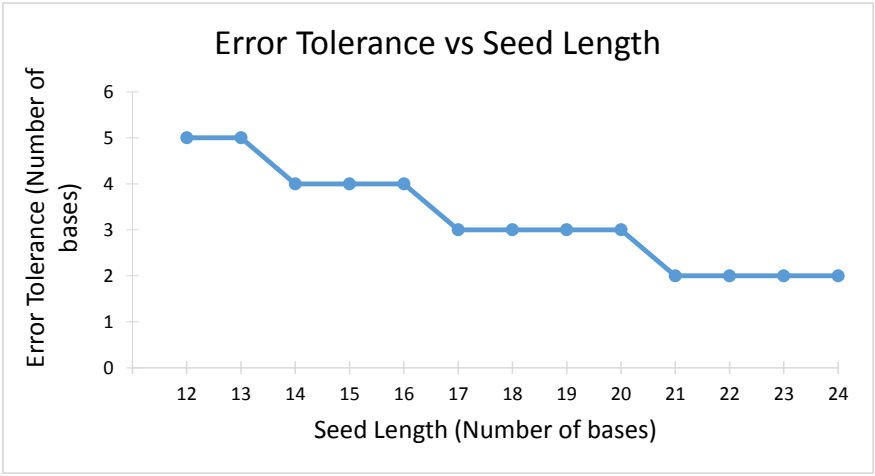


Figure 5: The maximal number of tolerable errors for a 80 bases read with different seed lengths.

Figure 5 show that for a 80-base-long read, the number of tolerated errors decreases with longer seeds.

Methods

As we described in the "Dilemma" section, each seed contributes to the mapping cost quadratically proportional to the number of its locations (Equation 2). As a result the few expensive seeds in the lookup table may drastically hinder the speed of the mapper, which coincides with their ubiquitous appearances in the genome. We observe that at seed length 12, the 1.08% expensive seeds in the lookup table, which have more

than 100 locations, account for 95.7% of the total mapping cost of all the seeds. While longer seeds reduces the mapping cost of the mapper, they also reduce the memory-efficiency of the mapper as well as the error tolerance of the mapper.

Our goal in this paper is to design a mechanism that provides small mapping cost while preserving high memory-efficiency and high error-tolerance.

With the above observation, we propose using different seed lengths for different seeds, according to the frequencies (number of locations) of the seed in the reference genome. In particular, we replace only the expensive short seeds with longer cheaper seeds while preserving the original cheap short seeds as is. Consequently, the seeds in the lookup table all become cheap seeds which have similar numbers of locations but nonuniform lengths. We call these seeds "Heterogeneous Seeds". Since heterogeneous seeds only replace expensive short seeds which account for only 1.08% of the short seeds, while most short seeds in the lookup table are unmodified, the added overhead of heterogeneous seeds is very small compared to short seeds.

In the remainder of this paper, we first propose the "Heterogeneous Lookup Table", a data structure that enables heterogeneous seeds. We then discuss the "Jigsaw Seeds" and "Overlapping Seeds" which is a seed dividing method that retains the same level of error tolerance of heterogeneous seed as with short seeds.

Heterogeneous seeds and heterogeneous lookup table

Using longer seeds can be considered as extending short seeds with more bases, i.e. adding bases trailing to the short seeds to make them longer. Extending short seeds generates more permutations. For each single base added to a short seed, there will be four new permutations as the extended base can be any of A, C, G and T. The four new permutations partition and inherit the locations of the original short seed, each taking a portion of it.

The seed extension can also be visualized in the form of a tree. Consider a tree where each node represents a seed at a certain length, as the tree in Figure 6, and its four children represent the extensions with one more base extended to it—A, C, G and T respectively, denoted by edge label. The root node of the tree is one of the short expensive seeds.

Previous works have used seeds of a uniform length throughout the mapping process (e.g seeds of length 12 in mrFAST, seeds of length 20 in SNAP). We call such seeds as **homogeneous seeds**. Using longer seeds can be viewed as extending homogeneous short seeds uniformly to homogeneous long seeds.

While extending both cheap seeds and expensive seeds generate the same number of new permutations, according to Equation 2, extending a cheap seed provides little reduction in mapping cost. As a result, we

conclude that only expensive seeds should be extended into longer seeds while the extension should stop as soon as the extended long seeds becomes cheap.

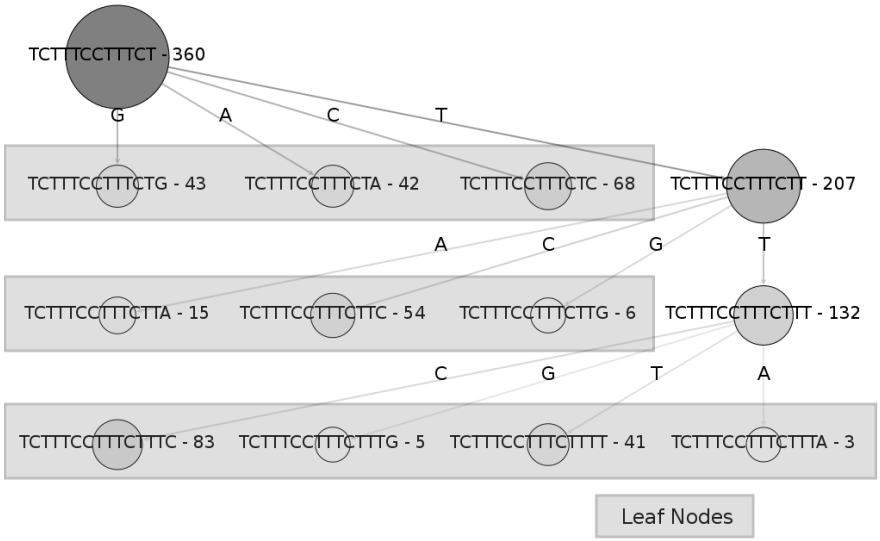


Figure 6: An extension tree of the expensive short seed "TCTTTCCTTTCT". Each node in the tree represents an extended longer seed of "TCTTTCCTTTCT". The number in each node represents the total number of locations in each seed. The leaf nodes in the extension tree are the seeds having fewer than 100 locations, i.e, the cheap seeds.

The extension strategy is best described as the extension tree shown in Figure 6. With a preset threshold τ, we define seeds with a number of locations greater than τ as **expensive seeds**, whereas seeds with a number of locations smaller than τ are **cheap seeds**. In this paper, we choose a threshold $\tau = 100$ (effects of choosing other thresholds are explored in the "Analysis" section). Figure 6 represents an extension tree of a basic seed "TCTTTCCTTTCT". As the figure shows, we keep extending the seed into longer seeds until the extended long seed comes a cheap seed, which is a leaf node of the tree. We stop further extending the seed when the seed becomes cheap since it is no longer cost efficient as we prove above. After we have extended the seed into all leaf nodes, we have reached a point where the expensive short seed has been efficiently extended into many long seeds at different lengths and the locations of the expensive short seeds are also evenly distributed among all long seeds.

With the help of the extension tree, we now revise the lookup table of homogeneous short seeds to enable extension for the expensive short seeds. We call this data structure "heterogeneous lookup table". The data structure is shown in Figure 7. On top of the original "seed $\rightarrow$ locations" permutation array, we add a key-value table, which is called **extension table**, to store the extensions of the expensive short seeds. The keys in the extension table are different extensions of an expensive short seed while the values

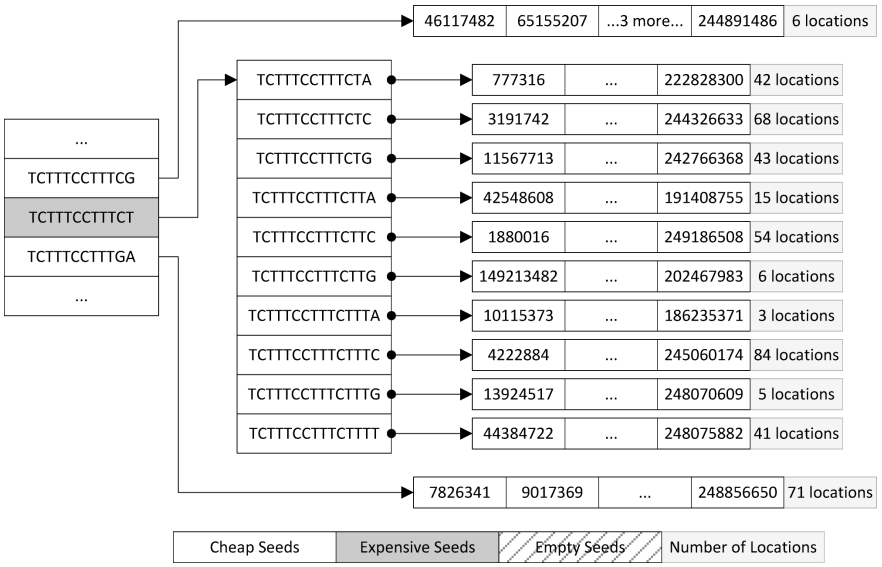


Figure 7: The heterogeneous lookup table. While the entries of cheap short seeds are left unmodified, the entries of expensive short seeds are extended into extension tables, which host extensions of the expensive seeds and their locations. Notice that no extended long seeds are expensive in the extension table (until the extension reaches the maximum length L_{max}).

are locations of the extended long seeds.

Ideally, we would like to keep extending the short seeds until all seeds are cheap. However, as we will describe later, allowing seeds to extend infinitely reduces the error-tolerance of the mapper. In this paper, we impose a maximum limit of seed length $L_{max} = 24$ (effects of choosing other maximum lengths are explored in the "Analysis" section). No seed in the heterogeneous lookup table is longer than this limit.

At mapping, a long seed with the length of L_{max} is sent to the lookup table for its locations. The seed is first divided into 2 sub-sequences: a head sequence with the same length of the short seeds which is called the "**basic part**" while the rest of the seed is called the "**extension part**". Basic part is used to query the permutation array. If the entry of the basic part is a cheap short seed, then no further extension is required and all locations of the cheap short seed is returned. If the entry of the basic part is an expensive short seed which has a pointer to an extension table in the permutation array, then the extension part is used to search for the potential matching in the extension table. Once the mapper found the matching extension, all locations of the extended long cheap seed are returned to the mapper.

Jigsaw seeds and overlapping seeds

With the heterogeneous seeds lookup table, the mapping costs of the seeds are drastically reduced with little increase in memory overhead. However, this solution is incomplete, as heterogeneous seeds require dividing the read into fewer long seeds (at the length of L_{max}), which yields poor error tolerance according to the pigeonhole principle.

If the read can be directly divided into cheap seeds at minimum lengths to query the heterogeneous lookup table (rather than L_{max}), or if the long seeds can overlap, as shown in Figure 8 and in Figure 9 respectively, then we can still extract many seeds from the read, regaining the high error tolerance. In this section, we propose "jigsaw seeds" and "overlapping seeds" for the two methods respectively.

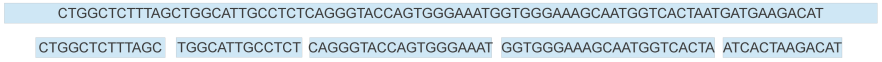


Figure 8: An example of jigsaw seeds: a read is divided into cheap seeds with different lengths.

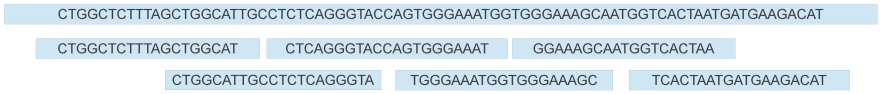


Figure 9: An example of overlapping seeds: a read is divided into long cheap seeds overlapping each other.

Jigsaw seeds: heterogeneous with higher error-tolerance

The main difficulty of proposing a static seed partition policy with high error-tolerance for heterogeneous seeds is that different seeds have different lengths and their placements in the read is highly read dependent. If dividing improperly, we may once again acquire expensive seeds. Luckily, with heterogeneous lookup table, it is unnecessary to have a static seed partition policy, instead, we propose a cheap dynamic seed partition policy that explores the minimal lengths of the cheap seeds to obtain the maximum number of seeds, which offers higher error-tolerance.

Instead of dividing the read into seeds in parallel, we propose a dynamic policy which divides the read into seeds in serial, as shown in Algorithm 1. While each time the mapper still uses a long seed (L_{max} bases) to query the heterogeneous lookup table, with the exact matching extension length returned by the lookup table, the mapper can reduce the length of the querying long seed (L_{max} bases) to the minimal length for the seed to remain cheap. The rest of the bases of the long seed is returned to the read to form the next seed. In this way, a read is partioned into as many non-overlapping cheap seeds as possible.

Algorithm 1: A dynamic seed partitioning algorithm that partitions the read into as many non-overlapping seeds as possible

Data: HET_LOOKUP_TABLE
Input: READ
Result: Obtain $e+1$ SEED
$i = 0$ $START = 0$ **while** $i < e+1$ **do**
    SEED = READ.substr(START, L_{max}) `// Obtain the long seed with the length` L_{max} `at the START position`

    LENGTH = HET_LOOKUP_TABLE.query(SEED) `// Get the exact length of the matching extension of the long seed`

    SEED = SEED.substr(0, LENGTH) `// Recalibrate the length of the seed to match the extended cheap long seed`

    START += LENGTH `// Increment the START position of the next long seed by LENGTH bases`
end

With the above algorithm, the mapper can obtain the maximum number of cheap seeds with different lengths jostling each other in the read. This is akin to a jigsaw puzzle, where each piece of the puzzle is all different but they interlock with each other, building up the entire picture. Therefore, we call this dynamic seed partition policy "Jigsaw Seeds".

Unlike hobbes [19], which needs many accesses to the lookup table in order to find the set of cheapest seeds, jigsaw seeds only need N accesses for N seeds and they guarantee that all seeds are as cheap as possible (before hitting the extension limit).

Overlapping seeds: a backup when jigsaw seeds fails

Jigsaw seeds works most of the time, since usually there are few long cheap seeds in a read. Nonetheless, there are still a few cases where a read contains multiple long seeds. For those reads, even partitioned with jigsaw seeds, there are still not enough seeds to tolerate all possible errors.

The problem originates from the fact that the pigeonhole principle requires the seeds to be non-overlapping. If the seeds are allowed to overlap each other slightly, where at any given point in the read, there are at most two seeds overlapping each other (Figure 9), then we can still partition the read into many seeds, even if there are multiple long cheap seeds. We call such partition policy as **overlapping seeds**. However, it is now possible that a single error simultaneously destroys seeds at once. As a result, e errors may destroy all $e+1$ overlapping seeds. Fortunately, as we will describe below, we can provide a

generalization of the pigeonhole principle by allowing seeds to have a single-base error in the extension part, restoring the high error-tolerance property of the mapper.

Theorem 1. *If a read contains e errors and is partitioned into $e+1$ overlapping seeds, where there are at most two seeds overlapping each other at any point in the read, then there must be at least one seed with no errors in its non-overlapping region and at most one error in its overlapping region.*

Proof. We prove by contradiction. Let us assume that such seed does not exist. Then each seed will have at least one error in its non-overlapping region or have at least two errors in its overlapping region. For the first case, if one or more errors is in the non-overlapping region of a seed, then no other seed shares the error, putting at least one error per seed. For the second case, since there are exactly two seeds sharing a single overlapping region, each seed inherits one half of the error. Consequently, at least two errors in the overlapping regions of two seeds is also equivalent to at least one error per seed of the two seeds. As a result, $e+1$ seeds will have at least $e+1$ errors, but we have only e errors, which is a contradiction.

□

Lemma 1. *With the same conditions as above, there must be a seed which not only has at most a single error in the overlapping region, but also the error locates specifically in its right overlapping region. In other words, there exists a seed which has no error in its left-overlapping region (if the seed overlaps with its left neighbor seed) or in its non-overlapping region.*

The proof of the Lemma is provided in the Appendix.

We further observe that in the heterogeneous lookup table, most expensive seeds have only a few extensions (12 on average with 3124 as the maximum). This observation suggests that instead of searching for perfect extensions of the long seed in the extension table, it is computationally inexpensive to search for extensions while allowing one error in the extension part of the seed. Although allowing errors forces the mapper to give up cheap searching algorithms (i.e., binary search), the additional computation is still much cheaper than simply using the short seeds. For example, for the expensive short seed that has the most extensions (3124), a linear search with one allowed error generates 3124 extension verifications (a small-scale verification for only the extension part of the seed), which returns one (or sometimes a few) cheap long seed with fewer than 100 locations. Compared using short seeds, where the same expensive short seed directly returns 244891486 locations to verify, the total computational cost is much less.

Moreover, unlike the verification calculation for the entire read, the extension verification need not to be 100% correct, since any extension passes the small-scale verification will go through a full read-long-scale

rigorous verification again later on. Extension verification only serves the purpose of a filtering mechanism. As long as the small-scale verification does not falsely reject correct extensions (as false negatives), it is acceptable for the verification to allow a few incorrect extensions (with more than one error) pass (as false positives).

We also propose a simple vectorization-friendly algorithm for extension verification, as shown in Algorithm 2, which runs very fast on modern processors that are capable of vector processing.

Algorithm 2: A vectorization-friendly extension verification algorithm that verifies if there exists at most 1 error in the extension part of the seed

---

Input: Two short sequences $SeedExt, TableExt$ `// As a running example, assume SeedExt=ACTACGTT, TableExt=ACTATCGT`

Output: A boolean if the edit distance is smaller or equal to 1 between the two sequences

$hamming\_mask \leftarrow SeedExt \oplus TableExt$`// Compute the bit-mask for the matching bases. For the running example, hamming_mask = 00001111; 0 represents a matching base while 1 represents a mismatching base`

$insertion\_mask \leftarrow (SeedExt \oplus (TableExt \gg 1)) \wedge hamming\_mask$ `// Compute the bit-mask for the matching bases with 1 insertion. For the running example insertion_mask = 00001111; 0 represents a matching base while 1 represents a mismatching base or an inserted base`

$deletion\_mask \leftarrow ((SeedExt \gg 1) \oplus TableExt) \wedge hamming\_mask$ `// Compute the bit-mask for the matching bases with 1 deletion. For the running example deletion_mask = 00001000; 0 represents a matching base while 1 represents a mismatching base or a deleted base`

`// Counting errors`

if $count\_ones(hamming\_mask) \leq 1$ **then**
  return *TRUE* `// Return true if there is none or only one mismatch.`

else if $count\_ones(insertion\_mask) \leq 1$ **then**
  return *TRUE* `// Return true if there is none or only one deletion.`

else if $count\_ones(deletion\_mask) \leq 1$ **then**
  return *TRUE* `// Return true if there is none or only one insertion. For the running example, TRUE is returned.`

else
  return *FALSE* `// Return false as there are more than one errors`

Notice that the above algorithm allows false negatives. For example, AAAACCAAAA is regarded

containing one error from AAAAATAAAA while in fact it contains two.

With the above observations, we complement the overlapping seed partition policy with a requirement of a minimum 12 bases (or the same length of the short seeds in the heterogeneous lookup table) non-overlapping and left-overlapping joint region for each overlapping seed. This additional requirement guarantees the existence of a seed which contains no error in its basic part and at most one error in its extension part.

With overlapping seeds, we approximate the error-threshold of short seeds with heterogeneous seeds.

Results

We simulated indexing human chromosome 1 with different configurations of seeds, including homogeneous seeds ranging from 12-base to 24-base, heterogeneous seeds with $\tau = 100$ and $L_{max} = 24$ and PatternHunter with weighted pattern "111010010100110111101" [20]. In this simulation, we assume all mappers use permutation arrays. We further assume each location list pointer in the permutation array occupies 4 bytes; each location in the location lists occupies 4 bytes and each entry in the extension table occupies 8 bytes (4 bytes for the extension, as each base occupies 2 bits, and 4 bytes for the location list pointer).

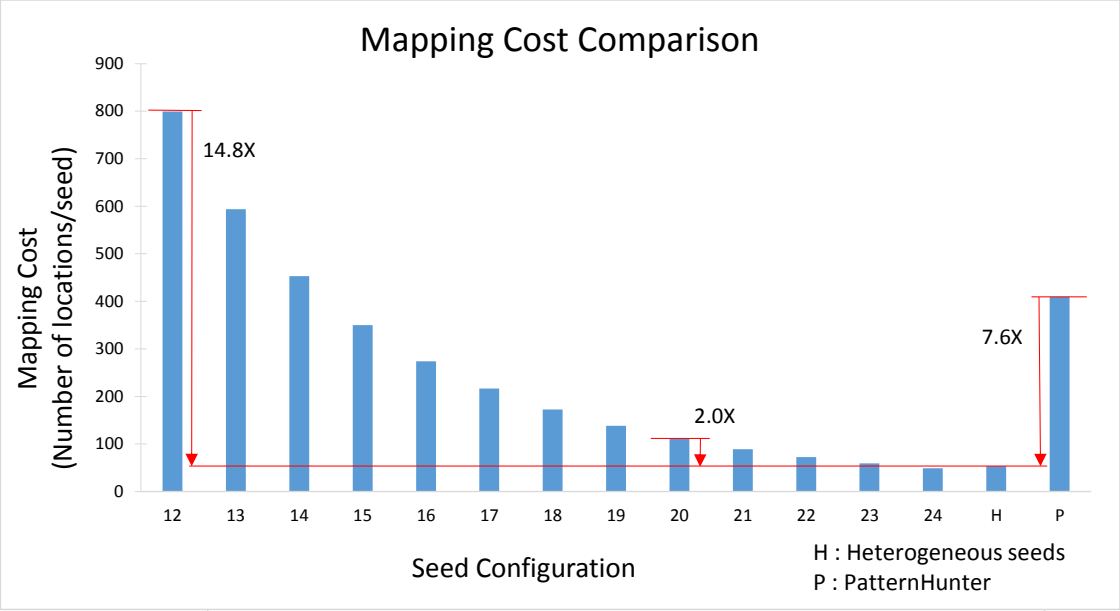


Figure 10: Mapping cost for different seed configurations for human chromosome 1.

Figure 10 shows the mapping cost of different seed configurations. Compared to short seeds (12-base), heterogeneous seeds reduce the mapping cost by 14.8x; while compared to long seeds (20-base), heterogeneous seeds reduce the mapping cost by 2.0x; Compared to PatternHunter, heterogeneous seeds

reduce the mapping cost by 7.6x. From this figure, we conclude that heterogeneous achieves and exceeds the reduction of computation of homogeneous long seeds.

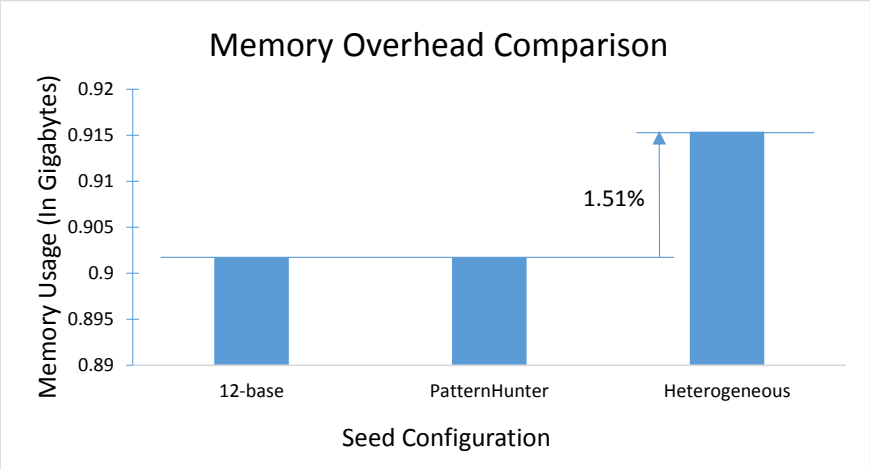


Figure 11: Memory consumption for different seed configurations for human chromosome 1.

Figure 11 shows the memory overhead of the data structures of homogeneous short seeds (12-base), heterogeneous seeds as well as PatternHunter. Since both homogeneous short seeds and PatternHunter use 12-base to generate the permutation array, they share the same memory overhead. In the case of heterogeneous seeds, only a slight increase of 1.5% in the memory consumption is observed, compared to homogeneous short seeds. For homogeneous long seeds (20-base), since very few long-seed mappers use full permutation array (which generates 4TB data), we only compare to the best case where only non-empty long seeds are stored. Under the best case, heterogeneous seeds still reduce the memory overhead of long seeds by 2x. In real implementations of using long seeds, however, mappers usually use partial permutation arrays which allow empty seeds. Assuming a 1:9 non-empty to empty seeds ratio (which is already very high, according to Figure 4), heterogeneous seeds provide 8.7x reduction in memory consumption.

We also simulated mapping 1 million 80-base long reads that are generated from the human chromosome 1 back to the same chromosome. Figure 12 presents the result of using short seeds (12-base) long seeds (20-base and 24-base), heterogeneous seeds and PatternHunter. From the graph, we observe that heterogeneous seeds on average reduce the number of locations to verify by 9.89x compared to short seeds (12-base), 1.29x compared to long seeds (20-base) and 4.73x compared to PatternHunter while increasing the number by 1.7x compared to 24-base long seeds. Moreover, hetergeneous seeds provide the same level of error-tolerance as short seeds whereas long seeds (20-base and 24-base) and PatternHunter cannot tolerate more than 3 errors (20-base can not tolerate more than 2 errors) hence their result are not shown beyond 3 errors.

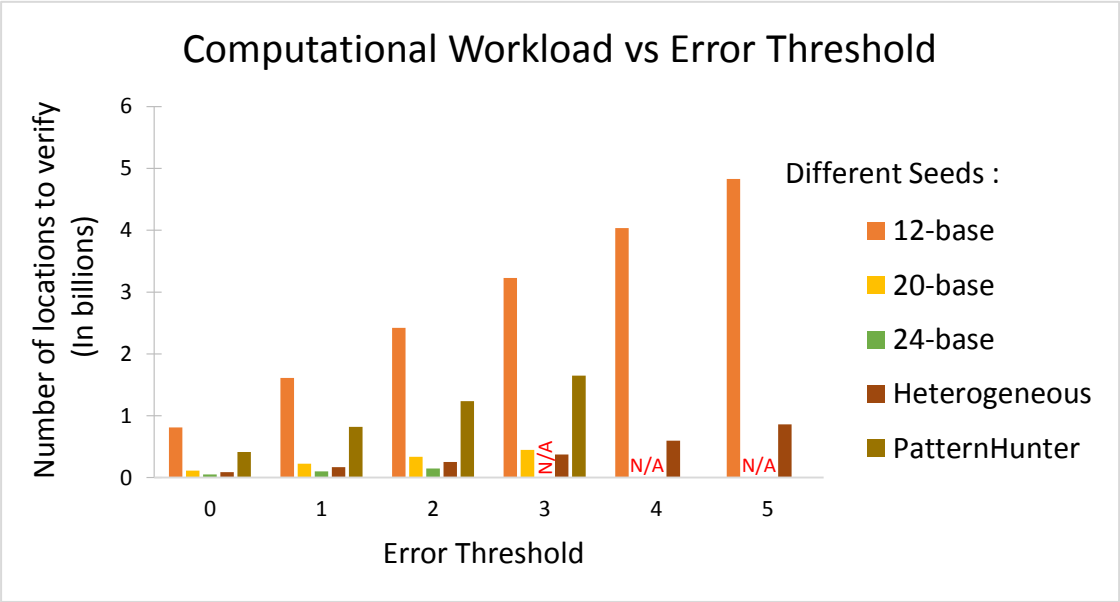


Figure 12: The number of locations to verify of different configurations of seeds to map 1 million 80-base read to human chromosome 1. Long seeds (20-base and 24-base) and PatternHunter do not support mapping with more than 3 errors (2 errors for 24-base seeds) for 80-base reads (this is indicated by N/A in the graph).

Analysis

In this section, we present the trade-offs of using heterogeneous seeds with different thresholds τ and different maximum lengths L_{max}.

With a smaller τ, more short seeds are tagged as expensive seeds and many long cheap seeds are further extended into even longer cheaper seeds. As a consequence, the memory consumption of heterogeneous seeds increases while the total mapping cost decreases as τ increases, as shown in Figure 13.

Similarly, with a longer maximum seed length L_{max}, long expensive seeds, seeds which still have more than τ locations at the length of original limit L_{max}, can be further extended into more longer seeds. As a consequence, the memory consumption of heterogeneous seeds increases while the total mapping cost of decreases as L_{max} increases, as shown in Figure 14.

As τ becomes smaller and L_{max} becomes longer, seeds also become longer and cheaper. As a result, a read can now be partitioned into fewer longer seeds. When high error-tolerance is desired, as more seeds are required, jigsaw seeds become less successful and overlapping seeds are used more frequently. This leads to fewer full-scale read verifications but more extension verifications, as Figure 15 shows. Although extension verifications involve much less computation than full-scale read verifications, with rapid growth, the increase in the computation of extensions verifications may still cancel out the benefit of the small reduction in the

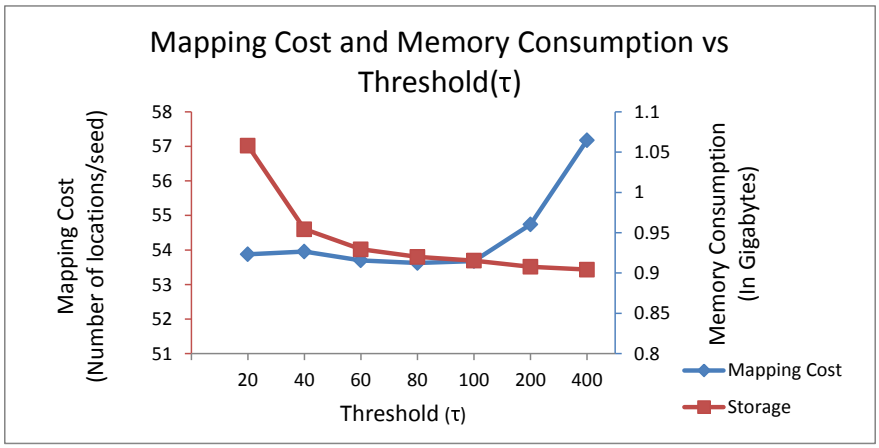


Figure 13: The reduction in mapping cost and the increase in memory consumption of the heterogeneous seeds with smaller threshold τ, while $L_{max} = 24$.

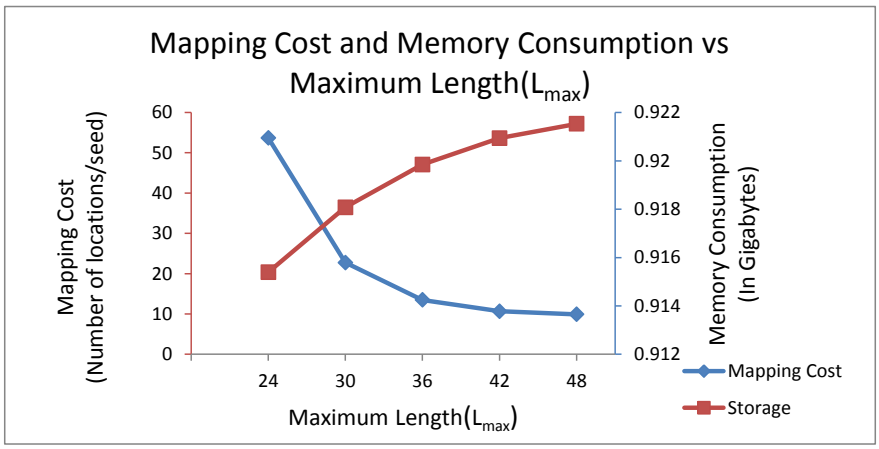


Figure 14: The reduction in mapping cost and the increase in memory consumption of the heterogeneous seeds with larger maximum seed length L_{max}, while $\tau = 100$.

number of read verifications.

While the optimal values of τ and L_{max} for heterogeneous seeds depend on the configuration of the computer, the desired error-tolerance of the mapper and the reads to be mapped, it is beyond the scope of this paper to formulate their relationships and to provide an optimal solution.

Conclusion

HTS platforms continue to evolve at a fast rate. New technologies are frequently introduced that offer different strengths; each, however, has unique biases. The current trend is to generate longer reads, with newer technologies such as the nanopore sequencing, at the cost of increased error rates.

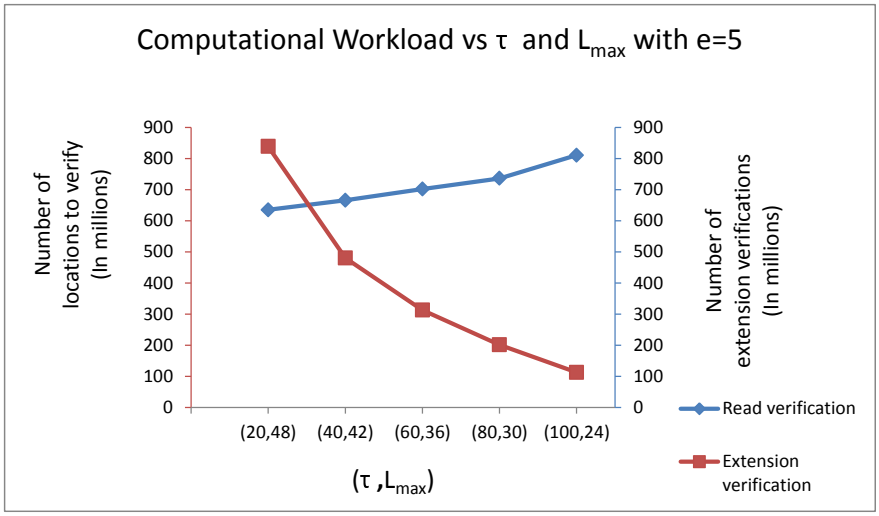


Figure 15: The reduction in the number of read verifications and the increase in the number of extension verifications with smaller τ and larger L_{max}, when mapping 1 million reads to human chromosome 1, under the requirement of tolerating 5 errors.

In this paper, we analyzed the effects of using different seed lengths on the speed, memory-efficiency and error-tolerance of conventional mappers. We observe that the few expensive short seeds in the lookup table severely reduce the speed of the mapper. Based on the analysis, we proposed "Heterogeneous Seeds" which simultaneously use seeds with different lengths in a mapper, according to the frequencies of the seeds in the reference genome.

With heterogeneous seeds, we approximate the massive reduction in computation provided by long seeds while at the same time retaining the high memory-efficiency and high error-tolerance provided by short seeds.

Author's contributions

HX designed the heterogeneous seeds. HX, HA and FH refined the idea. HX and HA performed benchmarks. OM and CA conceived and planned the experiments, and supervised HX for the algorithm development. All authors contributed to the writing of the manuscript.

Acknowledgments

This work was supported, in part, by an NIH grant HG006004 to C.A. and O.M., and a Marie Curie Career Integration Grant (PCIG10-GA-2011-303772) within the 7th European Community Framework Programme to C.A.

References

1. 1000 Genomes Project Consortium: **A map of human genome variation from population-scale sequencing.** *Nature* 2010, **467**:1061–1073.
2. Antonacci F, Kidd JM, Marques-Bonet T, et al.: **Characterization of six human disease-associated inversion polymorphisms.** *Hum Mol Genet* 2009, **18**:2555–2566.
3. Antonacci F, Kidd JM, Marques-Bonet T, et al.: **A large and complex structural polymorphism at 16p12.1 underlies microdeletion disease risk.** *Nat Genet* 2010, **42**:745–750.
4. Bailey JA, Eichler EE: **Primate segmental duplications: crucibles of evolution, diversity and disease.** *Nat Rev Genet* 2006, **7**:552–564.
5. Bailey JA, Gu Z, Clark RA, Reinert K, Samonte RV, Schwartz S, Adams MD, Myers EW, Li PW, Eichler EE: **Recent segmental duplications in the human genome.** *Science* 2002, **297**:1003–1007.
6. Bailey JA, Kidd JM, Eichler EE: **Human copy number polymorphic genes.** *Cytogenet Genome Res* 2008, **123**:234–243.
7. Bailey JA, Yavor AM, Massa HF, Trask BJ, Eichler EE: **Segmental duplications: organization and impact within the current human genome project assembly.** *Genome Res* 2001, **11**:1005–1017.
8. Bailey JA, Yavor AM, Viggiano L, Misceo D, Horvath JE, Archidiacono N, Schwartz S, Rocchi M, Eichler EE: **Human-specific duplication and mosaic transcripts: the recent paralogous structure of chromosome 22.** *Am J Hum Genet* 2002, **70**:83–100.
9. Bailey JA, Baertsch R, Kent WJ, Haussler D, Eichler EE: **Hotspots of mammalian chromosomal evolution.** *Genome Biol* 2004, **5**:R23.
10. Marques-Bonet T, Kidd JM, Ventura M, Graves TA, Cheng Z, Hillier LW, Jiang Z, Baker C, Malfavon-Borja R, Fulton LA, Alkan C, Aksay G, Girirajan S, Siswara P, Chen L, Cardone MF, Navarro A, Mardis ER, Wilson RK, Eichler EE: **A burst of segmental duplications in the genome of the African great ape ancestor.** *Nature* 2009, **457**:877–881.
11. Rozen S, Skaletsky H, Marszalek JD, Minx PJ, Cordum HS, Waterston RH, Wilson RK, Page DC: **Abundant gene conversion between arms of palindromes in human and ape Y chromosomes.** *Nature* 2003, **423**:873–876.
12. Scally A, Dutheil JY, Hillier LW, Jordan GE, Goodhead I, Herrero J, Hobolth A, Lappalainen T, Mailund T, Marques-Bonet T, McCarthy S, Montgomery SH, Schwalie PC, Tang YA, Ward MC, Xue Y, Yngvadottir B, Alkan C, Andersen LN, Ayub Q, Ball EV, Beal K, Bradley BJ, Chen Y, Clee CM, Fitzgerald S, Graves TA, Gu Y, Heath P, Heger A, et al.: **Insights into hominid evolution from the gorilla genome sequence.** *Nature* 2012, **483**:169–175.
13. Ventura M, Catacchio CR, Alkan C, Marques-Bonet T, Sajjadian S, Graves TA, Hormozdiari F, Navarro A, Malig M, Baker C, Lee C, Turner EH, Chen L, Kidd JM, Archidiacono N, Shendure J, Wilson RK, Eichler EE: **Gorilla genome structural variation reveals evolutionary parallelisms with chimpanzee.** *Genome Res* 2011, **21**:1640–1649.
14. Green RE, Krause J, Briggs AW, Maricic T, Stenzel U, Kircher M, Patterson N, Li H, Zhai W, Fritz MHY, Hansen NF, Durand EY, Malaspinas AS, Jensen JD, Marques-Bonet T, Alkan C, Prüfer K, Meyer M, Burbano HA, Good JM, Schultz R, Aximu-Petri A, Butthof A, Höber B, Höffner B, Siegemund M, Weihmann A, Nusbaum C, Lander ES, Russ C, et al.: **A draft sequence of the Neandertal genome.** *Science* 2010, **328**:710–722.
15. Reich D, Green RE, Kircher M, Krause J, Patterson N, Durand EY, Viola B, Briggs AW, Stenzel U, Johnson PLF, Maricic T, Good JM, Marques-Bonet T, Alkan C, Fu Q, Mallick S, Li H, Meyer M, Eichler EE, Stoneking M, Richards M, Talamo S, Shunkov MV, Derevianko AP, Hublin JJ, Kelso J, Slatkin M, Pääbo S: **Genetic history of an archaic hominin group from Denisova Cave in Siberia.** *Nature* 2010, **468**:1053–1060.
16. Burrows M, Wheeler DJ, Burrows M, Wheeler DJ: **A block-sorting lossless data compression algorithm** 1994.
17. Ferragina P, Manzini G, Mäkinen V, Navarro G: **Compressed representations of sequences and full-text indexes**. *ACM Transactions on Algorithms* 2007, **3**.
18. Xin H, Lee D, Hormozdiari F, Yedkar S, Mutlu O, Alkan C: **Accelerating read mapping with FastHASH**. *BMC Genomics* 2013, **14**(Suppl 1):S13, [http://www.biomedcentral.com/1471-2164/14/S1/S13].

19. Ahmadi A, Behm A, Honnalli N, Li C, Weng L, Xie X: **Hobbes: optimized gram-based methods for efficient read alignment**. *Nucleic Acids Research* 2011, **40**:e41.

20. Ma B, Tromp J, Li M: **PatternHunter: faster and more sensitive homology search**. *Bioinformatics* 2002, **18**:440–445.

21. Li H, Durbin R: **Fast and accurate long-read alignment with Burrows-Wheeler transform.** *Bioinformatics* 2010, **26**:589–595.

22. Matei Z, J BW, Kristal C, Armando F, David P, Scott S, Ion S, M KR, Taylor S: **Faster and More Accurate Sequence Alignment with SNAP**. *eprint arXiv* 2011.

Appendix
Related works

The seed length dilemma has concerned researches for a long time. Multiple data structures and mapping mechanisms attempted to resolve the dilemma yet they all fall short in certain aspects and failed to provide a efficient data structure and mechanism which occupies small amount of memory, provides fast accesses, generates very few locations to verify and at the same time tolerates many potential errors.

mrFAST and FastHASH: use short seeds for higher error tolerance

mrFAST [18] is a typical hash based mapper which uses short seeds (11-13 bases) for higher error tolerance. One of mrFAST unique features is guaranteeing finding all possible mappings of the read with up to as many errors as 8% of the entire read with relatively small memory footprint (around 4GB). As with other short seed mappers, mrFAST suffers from verifying too many locations provided by the short seeds which greatly reduces the speed of the mapper. FastHASH alleviate the burden of mrFAST by artificially selecting the seeds with fewer locations to query the lookup table in order to reduce the number of locations to verify. While this technique works fine when the number or errors is small, it does not work as well when the number of errors is large. When the number of errors is large, most of the seeds will be selected anyways, suggested by the Pigeonhole Principle, hence the selection of seeds becomes unnecessary.

BWA: use long seeds for fast perfect mapping

As its name suggests, BWA [21] uses Burrows-Wheeler transformation and is a typical BWT-FM mapper. Similar to all BWT-FM mappers, BWA uses long seeds (as long as the entire read); has relatively small memory footprint (4GB); and is very fast to find the perfect mapping. If there are errors in the read, BWA tries to fix the error by artificially altering the read base by base in a brute-force manner until it founds an acceptable mapping. Although BWA finds a perfect mapping very fast, when it is configured to find all

possible mappings of the read, it gets slowed down exponentially, because it has to search for all possible alternations of the read.

SNAP: use longer seeds for faster lookup

SNAP [22] is another hash based mapper which uses longer seeds (20 bases). Unlike BWA, SNAP do not use a BWT-FM or any complex lookup mechanism, rather, it uses a "sparse" permutation array for fast access with a hashing function which reduces the number of "empty seeds" in the table, saving memory space. Besides, SNAP filters out any seed having more than one locations, which further simplifies the data structure from a "seed $\rightarrow$ locations" map to a "location[seed]" array. With the simple lookup function and the long seed, SNAP achieves a speed much faster than BWA. Nonetheless, the fast speed of SNAP also come at a price. First of all, SNAP still consumes a lot of main memory. To map with human reference genome, SNAP requires 64 GB of main memory. Second of all, the long seeds and the heuristic filtering reduce the error tolerance of SNAP and cancel out the guarantee of finding all possible mappings provided by the "Pigeon-Hole" Principle.

Hobbes: find the best seed placement

Instead of changing the seed length, hobbes [19] proposes to search for the best seed placement which yields the fewest locations. By the Pigeonhole Principle, to tolerate e errors, a mapper only needs to verify the locations of $e+1$ non-overlapping seeds. The only requirement of the placement of the seeds is that the seeds do not overlap each other. There is no requirement about where should the seeds locate in the read. In fact, they can be anywhere: they can be contiguous or they can be discrete. Hobbes takes advantage of this property and list out all possible seed placements and select the placement yields the fewest locations to verify. However, the computation to evaluate all seed placements is non-trivial, since there can be roughly $O(\binom{L}{e+1})$ total possible placements (L is the length of the read and e is the number of allowed errors). Further more, in order to calculate the number or locations provided by all potential placements, hobbes needs to query the lookup table many times to get the number of locations for all possible seeds. Last but not the least, for some cases, a read contains a long repeating segment (more than 20 bases), which presents at many places in the genome. If that is the case, merely changing the placement of the seed do not help much, as the number of locations between two close-by seeds may be very similar.

PatternHunter: mimic long seeds—redistributes the locations to the seeds for fewer verifications

So far we have been assumed seeds are consist of continuous bases (e.g., ACTCATTACATC). However, this is not a requirement—seeds can also be made up of intermittent bases (e.g., A_T_AT_A_AT_C_TACG from the continuous sequence ACTCATTACATCCATACG). As long as seeds do not overlap or interleave each other, we can still use the Pigeonhole Principle for intermittent seeds (if two intermittent seeds interleave but not overlap each other, a single insertion or deletion can destroy both of them, violates the "one error destroys one seed" principle). Essentially, seeds with intermittent bases can be thought as using fewer bases (as many as a short seed) to mimic a long seed, because they are treated exactly like a long seeds throughout the mapping process except that they occupies less memory space. For example, during read division, a read is first divided into "long seeds" and afterwards, only when the mapper uses the "long seeds" to query the lookup table, the mapper extracts the intermittent bases from the "long seed" (ACTCATTACATCCATACG) to form the intermittent seed (A_T_AT_A_AT_C_TACG).

Intermittent seeds provides many benefits. One of the main benefits is a more balanced location distribution among all seeds. With the freedom to select which bases to take from the long seed, PatternHunter finds the best pattern which redistributes the locations the most evenly to the seeds (e.g., for human genome, the pattern of taking 12 bases from a 20-base long seed would be 11101001010011011101. "1" is taking and "0" is not taking). Compared to short seeds, PatternHunter does not increase the memory size as it uses as many bases as the short seeds to form the permutation array.

Still, PatternHunter has several limitations. First of all, PatternHunter divides the read into long seeds, which implies the error tolerance of it will be weaker than short-seed mappers. More importantly, its power of redistribution is limited. PatternHunter only provides a universal pattern that is applied to all of the seeds. However, based on our observation, the seeds contribute the most to the mapping cost are the expensive seeds. PatternHunter does not focus on reducing the mapping cost of the expensive seeds, rather it selects the "best pattern" which reduces the total mapping cost for all seeds, in the hope that it will reduce the mapping cost of the expensive seeds as well, which is not guaranteed. Sometimes a pattern that reduces the mapping cost of some expensive seeds will increase the mapping cost of the other expensive seeds, limiting its effectiveness.

Proof of Lemma 1

Proof. We again prove by contradiction. Let us assume such seed does not exist, then each seed will either have at least one error in the non-overlapping region or have at least have two errors in the overlapping

region or have a single error in the left overlapping region.

According to Theorem 1, there must exist at least one seed with an intact non-overlapping region and a single error in the left overlapping region. A seed having an error in its left overlapping region also implies that its left neighbor seed has an error in the right overlapping region. Since there exists no seed that only contains a single error in the right overlapping region as we assumed, the left neighbor seed has to have at least one error either in the non-overlapping region or in the left overlapping region. If the error is in the non-overlapping region, then we have formed a **complete chain**, which is defined as a chain of seeds which is linked together by errors in the common overlapping regions. A complete chain links as many as seeds as possible until it reaches to a seed who has no error in the overlapping region and breaks the link. If the error is in the left overlapping region, however, the two seeds will not form a complete chain, as the error in the left overlapping region links yet another seed. The link continues until it either reaches to a seed which has no error in the left overlapping region of that seed or it reaches the left most seed in the read which has no left overlapping region at all. However, as the chain grows, each time when it adds one more seed, it also includes at least one more errors, resulting at least one error per seed in the chain.

By definition, any seed without error in the right overlapping region will be the right end of a chain. Likewise, any seed without error in the left overlapping region will be the left end of a chain. A seed with no error in the overlapping region will be a complete chain by itself. Seeds without overlapping regions, such as the left most and the right most seeds of the read, will also be the ends of complete chains. Each complete chain can have at most a single left seed end and a single right end seed. The left end of a complete chain may freely overlaps with the right end of another complete chain in the read as long as they do not share errors at the overlapping region.

With our assumptions, a read can only contain two kinds of chains: the first kind contains seeds which either has at least an error in the non-overlapping region or two errors in the overlapping regions as we described in Theorem 1; and the second kind is the same with the first kind except that the right most seed of the chain only has a single error in the left overlapping region. Nonetheless, both kinds of chains require at least one error per seed, requiring $e + 1$ errors for $e + 1$ seeds, contradicts to our given condition.

□