# Hobbes3: Dynamic Generation of Variable-Length Signatures for Efficient Approximate Subsequence Mappings

Jongik Kim
Division of Computer Science and Engineering
Chonbuk National University
Jeonju, Republic of Korea
Email: jongik@jbnu.ac.kr

Chen Li
Department of Computer Science
University of California, Irvine, USA
Email: chenli@ics.uci.edu

Xiaohui Xie
Department of Computer Science
University of California, Irvine, USA
Email: xhx@ics.uci.edu

***Abstract*—Recent advances in DNA sequencing have enabled a flood of sequencing-based applications for studying biology and medicine. A key requirement of these applications is to rapidly and accurately map DNA subsequences to a reference genome. This DNA subsequence mapping problem shares core technical challenges with the similarity query processing problem studied in the database research literature. To solve this problem, existing techniques first extract signatures from a query, then retrieve candidate mapping positions from an index using the extracted signatures, and finally verify the candidate positions. The efficiency of these techniques depends critically on signatures selected from queries, while signature selection relies on an indexing scheme of a reference genome. The $q$-gram inverted indexing, one of the most widely used indexing schemes, can discover candidate positions quickly, but has the limitation that signatures of queries are restricted to fixed-length $q$-grams. To address the problem, we propose a flexible way to generate variable-length signatures using a fixed-length $q$-gram index. The proposed technique groups a few $q$-grams into a variable-length signature, and generates candidate positions for the variable-length signature using the inverted lists of the $q$-grams. We also propose a novel dynamic programming algorithm to balance between the filtering power of signatures and the overhead of generating candidate positions for the signatures. Through extensive experiments on both simulated and real genomic data, we show that our technique substantially improves the performance of read mapping in terms of both mapping speed and accuracy.**

## I. Introduction

Finding similar objects is an important operation used in a wide range of applications. In this paper, we focus on the problem of finding positions of DNA subsequences in a large reference sequence approximately, which is known as the *read mapping* problem in genomics. Although the problem has been actively studied in the field of computational biology, it shares core technical challenges with similarity query processing studied in the database research literature.

Informally, given a query that is a DNA subsequence called a *read*, the read mapping problem is to find all positions of subsequences in a reference genome sequence that are similar to the query. The similarity between two sequences is measured using their edit distance, and a threshold is given to specify the number of acceptable errors and genetic variations. Figure 1 shows an example of a read mapping. In the figure, the mapping positions of the read AACT are 104, 112, and 116

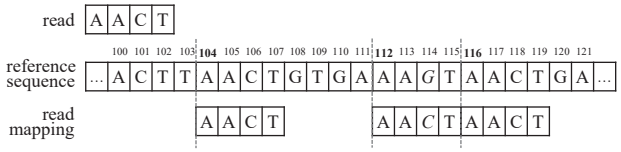


Fig. 1. An example of a read mapping

of the reference sequence where the read occurs either exactly or approximately.

To solve the problem, many techniques have been proposed under a filtering-and-verification framework, where candidate positions are generated using one or more filters and then verified to find true mapping positions. A majority of the effort has been focused on generating as few candidates as possible in an early stage of the read mapping because the performance of verification relies on the number of candidates. To generate candidate positions, existing techniques first extract signatures from a query, and then identify positions containing some signatures of the query as candidate positions. Many techniques make use of $q$-grams as signatures, where a $q$-gram of a sequence $s$ is a subsequence of $s$ of length $q$. They build a $q$-gram inverted index on a reference genome sequence, where each distinct $q$-gram in the reference is associated with an inverted list containing positions where the $q$-gram appears. Given an inverted index, a typical approach to generating candidates is to extract some $q$-grams from a read, retrieve inverted lists of the $q$-grams from the index, and take the union of positions on the inverted lists considering relative positions of the $q$-grams in the read (see Section II for details).

Since signatures of a query determine the filtering power, selecting an optimal set of signatures is an actively studied problem in similarity query processing as well as read mapping. Given a $q$-gram inverted index, for example, Hobbes [1] and Hobbes2 [2] proposed techniques for selecting an optimized combination of $q$-grams of a read to minimize candidate mapping positions. Because the length of signatures of a read is fixed to $q$, however, some $q$-grams might generate many candidate positions, which leads to inefficient mapping. There are several similarity query processing techniques that make use of variable-length signatures. VGRAM [3], [4] extracts some variable-length grams from data strings and makes an inverted

index for the extracted grams. VGRAM selects signatures of a query among those grams extracted for indexing. PASS-JOIN [5] and HSTree [6] build a separate inverted index for each distinct length of data strings by partitioning data strings into equally sized segments. Given a query, they generate signatures of the query according to the lengths of indexed segments. As these techniques rely on a predefined set of signatures, which are chosen regardless of queries, however, some queries could suffer from poor performance caused by inappropriate signatures. Hence, we need a dynamic way of adjusting signature lengths to select good signatures for each query.

To address the problem, we propose a method to produce variable-length signatures from a read using a fixed-length $q$-gram inverted index. We show that we can generate candidate positions for a variable-length signature using inverted lists of a few fixed-length $q$-grams. We also propose a novel algorithm to judiciously select optimized signatures for each query. The following are the main contributions of the paper.

- We have observed that overlapping and consecutive $q$-grams can be combined into variable-length signatures. Based on the observation, we generate candidate positions for a variable-length signature using inverted lists of fixed-length $q$-grams.
- We propose a novel dynamic programming algorithm that judiciously selects variable-length signatures by balancing between the filtering power of the selected signatures and the overhead of generating candidates for the signatures.
- We have developed Hobbes3, a software package for mapping reads based on the proposed technique and made it available online.[1]
- We have conducted a thorough experimental study using real genomic data and compared Hobbes3 against state-of-the art read mapping softwares.

The rest of the paper is organized as follows. Section II provides background on subsequence mappings and related work. Section III presents the proposed technique for generating and selecting signatures. Section IV provides our candidate generation and verification methods. Section V presents experimental results and Section VI concludes the paper.

## II. Preliminaries and related work

### A. Read Mapping Problem

A genome is a sequence of characters from the alphabet $\Sigma = \{\mathtt{A}, \mathtt{C}, \mathtt{G}, \mathtt{T}, \mathtt{N}\}$. The following two measures are used to compute the similarity between two genomic sequences.

*Hamming distance:* given two sequences of $s_1$ and $s_2$ with the same length (i.e., $|s_1| = |s_2|$), the hamming distance between them is the number of positions $i$ such that $s_1[i] \neq s_2[i]$, where $|s|$ denotes the length of a sequence $s$ and $s[i]$ denotes the $i^{th}$ character of $s$.

*Edit distance:* given two sequences of $s_1$ and $s_2$ whose lengths are not necessarily the same, the edit distance between them is the minimum number of edit operations to transform $s_1$ to $s_2$, where an edit operation is substitution, insertion, or deletion of a single character.

*Example 1:* Consider two sequences $s_1 = \mathtt{AACCGGT}$ and $s_2 = \mathtt{ACCGGTA}$. The hamming distance between them is 4 since characters at position 1, 3, 5, and 6 are different (assuming positions starting from 0). Their edit distance is 2 because $s_1$ can be transformed to $s_2$ by deleting *A* at position 0 and inserting *A* at the end.

Given a set of reads $R$ and a similarity threshold for a similarity measure, the read mapping problem is to find every position $l$ of a reference sequence $\Gamma$ for each read $r \in R$ such that the similarity between $r$ and a subsequence of $\Gamma$ starting at $l$ is within the threshold.

Edit distance is an extended measure of hamming distance since it supports *indel* (insertion/deletion) errors in addition to mismatch (or substitution) errors. As DNA sequencing technologies are progressing toward producing longer reads, it is important to support indel errors, which are caused by sequencing errors and genetic variations [2]. Hence, we use the edit distance to measure similarities between sequences in this paper. Because Hobbes3 directly inherits the functionality of read mapping using hamming distance from Hobbes2, interested readers may refer to [2] for the comparison results of read mapping using hamming distance. For the ease of description, we first focus on the read mapping problem with mismatch errors only. Then we give a separate discussion of how to support indel errors in Section IV.

### B. Candidate Generation Using q-gram Signatures

A positional $q$-gram of a sequence $r$ is a subsequence of $r$ of length $q$ with the start position of the subsequence in $r$. The $q$-gram set of $r$, denoted by $G(r, q)$, is obtained by sliding a window of length $q$ over $r$.

*Example 2:* For a sequence $r = \mathtt{GAATGAAT}$, $G(r, 3)$ is $\{(\mathtt{GAA}, 0), (\mathtt{AAT}, 1), (\mathtt{ATG}, 2), (\mathtt{TGA}, 3), (\mathtt{GAA}, 4), (\mathtt{AAT}, 5)\}$.

As a character in a sequence $s$ is included in at most $q$ positional $q$-grams, one substitution on $s$ modifies at most $q$ positional $q$-grams of $G(s, q)$. Therefore, if a sequence $s$ is different from another sequence $r$ by $k$ substitutions, $s$ should share at least $T = |G(s, q)| - k \times q$ common $q$-grams with $r$, where $|G(s, q)|$ denotes the size of $G(s, q)$. By counting the number of common $q$-grams between sequences, we can safely filter out dissimilar sequences without calculating their real distance. This technique is known as count filtering.

Besides count filtering, there is another technique widely used to filter out dissimilar sequences. The technique makes use of non-overlapping $q$-grams based on the following observation. Given a sequence $r$ and a hamming distance threshold $k$, we select $k + 1$ non-overlapping $q$-grams of $r$. Since each substitution can modify at most one of the $k + 1$ non-overlapping $q$-grams, the pigeonhole principle guarantees that it is impossible to distribute $k$ substitutions into $k + 1$ non-overlapping $q$-grams without leaving at least one $q$-gram unchanged. Therefore, if a sequence $s$ is different from $r$ by $k$ substitutions, $s$ should contain at least one $q$-gram of the selected $k + 1$ non-overlapping $q$-grams of $r$. This technique can be generalized by selecting $k+c$ non-overlapping $q$-grams

[1] http://hobbes.ics.uci.edu

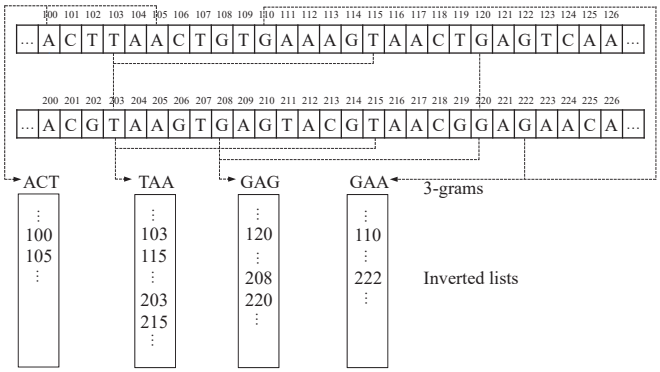


Fig. 2. Excerpt of a reference and a portion of its 3-gram inverted index

of $r$. In this case, $s$ should contain at least $c$ $q$-grams among the selected $k + c$ $q$-grams.

These filtering techniques are often used with an inverted index. We first retrieve inverted lists of $q$-gram signatures extracted from a read, and then generate candidate positions as follows. If we use the count filtering, we select those positions as candidate positions that appear on at least $T$ inverted lists. If we use the pigeonhole principle, we take the union of the inverted lists to generate candidate positions. When using these techniques, we should normalize positions on an inverted list considering relative positions of the corresponding $q$-gram in a read. That is, if the position of a signature $q$-gram in a read is $l$, we subtract $l$ from each position $p$ on the inverted list of the $q$-gram in order to align $p$ with the start position of the read. The following example illustrates candidate generation using the pigeonhole principle.

*Example 3:* Consider a read $r =$ TAACTGAGAAATTA and a reference sequence with its 3-gram inverted index shown in Figure 2. Given a hamming distance threshold 2, we can generate candidate positions using three non-overlapping 3-grams from $r$ based on the pigeonhole principle. Suppose we select (TAA, 0), (GAG, 5), and (TTA, 11) in $G(r, 3)$. We first look up the selected 3-grams in the index and retrieve two inverted lists of TAA and GAG (note that the 3-gram TTA does not appear in any position of the reference sequence, and thus the index does not contain an inverted list for the gram). We then take the union of the two inverted lists as follows. We directly add positions on the inverted list of TAA to a candidate list since these positions are aligned with the read's start position. For the positions on the GAG's inverted list, we normalize them to align the read's start position. The 3-gram GAG appears at position 5 in the read, so the normalized positions of the elements on the GAG's inverted list are $120 - 5 = 115$, $208 - 5 = 203$, and $220 - 5 = 215$, respectively. We merge the normalized positions with the candidate list to generate candidates $\{103, 115, 203, 215\}$. By verifying subsequences of the reference starting at these positions, we can get a mapping at position 103 with a hamming distance 2.

### C. Related Work

Many techniques have been proposed for efficient processing of similarity queries [3], [4], [5], [6], [7], [8], [9], [10], [11]. The focus of these techniques has been on reducing the number of candidates in a filtering phase. Most of them make use of inverted index to generate candidates. Early work (e.g., [8], [9], [10], [11]) utilizes variations of the count filtering, while recent techniques (e.g., [5], [6], [7]) have been proposed based on the pigeonhole principle.

There are a number of techniques proposed for solving read mapping problems. Hobbes [1] is a software package proposed to identify all mapping positions of a read. It generates candidate positions using inverted lists of non-overlapping $q$-grams with the help of bit vectors. Hobbes2 [2] uses additional prefix $q$-grams instead of bit vectors to further reduce the number of candidates initially generated. Recently developed read mappers (e.g., RazerS3 [12] and Masai [13]) have focused on supporting indel errors and improved the performance in terms of accuracy and speed. RazerS3 generates accurate mapping results by controlling mapping sensitivity based on its error-estimation technique. Masai reduces mapping time significantly by building an index on input reads and simultaneously generating candidate positions for multiple reads. More recently, Bitmapper [14] has been proposed for improving candidate verification. It verifies multiple candidate positions at the same time using a variation of Myers' bit vector verification algorithm [15].

Other popular read mapping softwares (e.g., Bowtie [16], Bowtie2 [17], BWA [18], and Gem [19]) have been developed aiming at identifying one or a few top mapping positions for each read. This mapping strategy leads to a significant improvement in mapping speed. However, in many applications such as ChIP-seq experiments [20] and RNA-seq transcript abundance quantification [21], [22], it is often more desirable to identify *all* candidate positions of reads.

## III. Variable-Length Signature Selection

In this section, we first propose our technique to make a variable-length signature and generate candidate positions of the signature using fixed-length $q$-gram inverted index. We then present a dynamic programming algorithm that selects optimal signatures using a cost model.

### A. Generating Variable-Length Signatures

We begin with a notation and basic definitions useful for describing the technique. For a sequence $r$, let $s[b \,..\, e]$ denote the subsequence of $s$ that begins at position $b$ and ends at position $e$. We call a subsequence of $s$ a *segment*. Since we use a subsequence of a read as a signature, we use a signature and a segment interchangeably in this paper.

*Definition 1:* Given a reference sequence $\Gamma$ and a segment $s$ of a read occurring at a position $l$, $L(s)$ denotes a set containing every position $o$ in $\Gamma$ such that $\Gamma[o+l \,..\, o+l+|s|-1] = s$. Let $I(s)$ be an inverted list of $s$. Then, $L(s) = \{p-l | p \in I(s)\}$.

*Example 4:* Consider the reference sequence in Figure 2 and a 3-gram (GAG, 5) of the read $r$ in Example 3. $L(\texttt{GAG})$ is $\{115, 203, 215\}$, which contains normalized positions of the elements on the GAG's inverted list.

*Definition 2:* Given two overlapping or consecutive segments $s_1$ and $s_2$ of a read $r$ occurring at positions $l_1$ and $l_2$ respectively, $s_1 \cdot s_2$ denotes a segment $r[l_1 \,..\, l_2 - l_1 + |s_2| - 1]$.

*Example 5:* For the two overlapping 3-grams $g_1 = (\texttt{TTA}, 0)$ and $g_2 = (\texttt{ACT}, 2)$ of the read $r$ in Example 3, $g_1 \cdot g_2$

denotes the positional 5-gram (TTACT, 0) of $r$. For the two consecutive 3-grams $g_1 = (\texttt{TTA}, 0)$ and $g_3 = (\texttt{CTG}, 3)$, $g_1 \cdot g_3$ denotes the positional 6-gram (TTACTG, 0) of $r$.

*Lemma 1:* Given a reference sequence $\Gamma$ and two overlapping or consecutive segments $s_1$ and $s_2$ of a read $r$, $L(s_1) \cap L(s_2) = L(s_1 \cdot s_2)$.

*Proof:* Let $l_1$ and $l_2$ be a position of $s_1$ and $s_2$ in $r$, respectively. The position of $s_1 \cdot s_2$ in $r$ is $l_1$ (by Definition 2). For each position $l$ in $\Gamma$ where $s_1 \cdot s_2$ appears, $l - l_1$ is contained in $L(s_1 \cdot s_2)$ (by Definition 1). We first show that $l - l_1$ is also contained in both $L(s_1)$ and $L(s_2)$ to prove $L(s_1 \cdot s_2) \subset L(s_1) \cap L(s_2)$. Since the position of $s_1 \cdot s_2$ in $\Gamma$ is $l$, $s_1$ appears at position $l$ in $\Gamma$ and $s_2$ appears at position $l + l_2 - l_1$ in $\Gamma$ by Definition 2. Hence, the normalized position of $s_1$ is $l - l_1$, and that of $s_2$ is $(l + l_2 - l_1) - l_2 = l - l_1$. By Definition 1, these positions are contained in $L(s_1)$ and $L(s_2)$, respectively.

For each position $l' \in L(s_1) \cap L(s_2)$, we now show that $l' \in L(s_1 \cdot s_2)$ to prove $L(s_1) \cap L(s_2) \subset L(s_1 \cdot s_2)$. Since $l' \in L(s_1) \cap L(s_2)$, $s_1$ appears at position $l' + l_1$ in $\Gamma$ and $s_2$ appears at position $l' + l_2$ in $\Gamma$ by Definition 1. Hence, $s_1 \cdot s_2$ appears at position $l' + l_1$ in $\Gamma$ and $l' \in L(s_1 \cdot s_2)$. ■

Given a $q$-gram inverted index of a reference sequence, we can combine a few overlapping and/or consecutive $q$-grams in a read into a variable-length signature and generate positions of subsequences in the reference that contain the signature at the same relative position as the read. Hence, we can generate candidate mapping positions for a read by selecting fixed-length $q$-grams, combining them into $k+1$ non-overlapping variable-length signatures, and taking the union of positions in the reference corresponding to these signatures. After we define a *positional intersection*, we give an example to demonstrate how we generate candidate positions using variable-length signatures extracted from a read.

*Definition 3:* Given two sets of positions $S_1$ and $S_2$, their *positional intersection*, denoted by $S_1 \cap_{\Delta n} S_2$, is defined as $\{l_1 | l_1 \in S_1 \; and \; \exists \; l_2 \in S_2 \; s.t. \; l_2 - l_1 = n\}$.

*Example 6:* In Example 3, suppose we choose three segments of (TAACT, 0), (GAGAA, 5), and (TTA, 11) as signatures from the read. By the pigeonhole principle, candidate mapping positions are contained in $L(\texttt{TAACT}) \cup L(\texttt{GAGAA}) \cup L(\texttt{TTA})$. In order to generate candidate positions for the segment TAACT, we decompose the 5-gram (TAACT, 0) into the two overlapping 3-grams of (TAA, 0) and (ACT, 2) since we have a 3-gram inverted index. By Lemma 1, $L(\texttt{TAACT}) = L(\texttt{TAA}) \cap L(\texttt{ACT})$. For each position $l \in L(\texttt{TAA}) \cap L(\texttt{ACT})$, $l_1 = l + 0 \in I(\texttt{TAA})$ and $l_2 = l + 2 \in I(\texttt{ACT})$ (by Definition 1). That is, for each position $l_1 \in I(\texttt{TAA})$ and each position $l_2 \in I(\texttt{ACT})$, $l_1 - 0 \in L(\texttt{TAACT})$ if and only if $l_2 - l_1 = 2$, where 2 is the difference of positions between the two 3-grams of (TAA, 0) and (ACT, 2). Therefore, the positional intersection between $I(\texttt{TAA})$ and $I(\texttt{ACT})$ can be used to generate $L(\texttt{TAACT})$ as follows:

$$L(\texttt{TAACT}) = \{l - 0 | l \in I(\texttt{TAA}) \cap_{\Delta 2} I(\texttt{ACT})\} = \{103\}.$$

Likewise, we generate $L(\texttt{GAGAA})$ and $L(\texttt{TAA})$ using the inverted lists of the 3-grams, GAG, GAA, and TAA, as follows:

$$L(\texttt{GAGAA}) = \{l - 5 | l \in I(\texttt{GAG}) \cap_{\Delta 2} I(\texttt{GAA})\} = \{215\},$$
$$L(\texttt{TTA}) = \{l - 11 | l \in I(\texttt{TTA})\} = \emptyset.$$

Hence, the candidate mapping positions are 103 and 215. By verifying the positions, we get a true mapping position of 103 with a hamming distance 2.

### B. Optimal Signature Selection

Selecting a good combination of signatures is crucial to the performance of read mapping. As we describe in Section II, we need at least $k + 1$ non-overlapping signatures to generate candidate positions for a hamming distance threshold $k$. Hobbes [1] selects $k + 1$ $q$-grams such that the sum of frequencies of the selected $q$-grams is minimized. Its approach is an optimized one in that both the verification cost for candidate positions and the scanning cost for inverted lists are minimized. Unlike Hobbes, however, selecting signatures by minimizing the sum of their frequencies does not guarantee achieving the best performance in our technique. Let us consider the following example.

*Example 7:* Given a reference sequence, Figure 3 shows a read and $q$-gram frequencies at each position of the read (i.e., the number of positions in the reference sequence that each $q$-gram appears). Suppose we have a 4-gram inverted index of the reference sequence. Given a hamming distance threshold 2, Hobbes selects three non-overlapping 4-grams of (TCTC, 2), (ACCC, 6), and (GAAC, 11) (circled in the figure) to minimize the sum of frequencies of selected grams. Using the proposed technique described in the previous section, we may select three variable-length signatures (GGTCT, 0), (CACCCT, 5), and (GAAC, 11) (pentagon-shaped in the figure). Note that their sum of frequencies is minimum among those of all possible three non-overlapping segments of the read. To generate candidate positions for the 6-gram of (CACCCT, 5), however, we need to scan a high-frequency 4-gram of (CACC, 5). Hence, the performance may not be optimized with the selected signatures although we can expect to minimize the number of candidates.

Since we dynamically generate positions in a reference sequence corresponding to a signature using $q$-gram inverted lists, the frequency of the signature is not proportional to the scanning cost of inverted lists as we illustrated in Example 7. Therefore, we need to balance between the number of candidates (or the cost of verifying candidates) and the cost of scanning inverted lists when we select variable-length signatures produced using the proposed technique. For this end, we need to estimate mapping cost (i.e., cost for generating and verifying candidates) of each combination of signatures. A naive approach is to enumerate all possible combinations of $k + 1$ signatures, evaluate their estimated costs, and choose

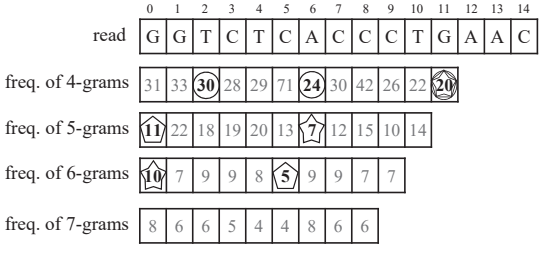


Fig. 3. A read and its frequencies of $q$-grams in a reference sequence

one with a minimum cost. However, enumerating all possible combinations is obviously inefficient.

We propose a dynamic programming algorithm that selects an optimized combination of signatures based on estimated costs. We assume that each signature generates candidates independently as all existing techniques do. Hence, we estimate the cost of mapping a read by summing up the mapping cost of each signature.

Given a $q$-gram inverted index of a reference sequence, the mapping cost of a signature $s$ of a read $r$ is estimated as follows. Let $Q$ be a set of $q$-grams used for making the segment $s$ as we described in the previous section. To generate candidates for $s$, we need to take the positional intersection of inverted lists of $q$-grams in $Q$. Hence, the cost for generating candidates is $\sum_{g \in Q} |I(g)|$, where $|I(g)|$ denotes the size of the inverted list of a $q$-gram $g$, or the frequency of $g$. To verify a candidate position for a hamming distance threshold, we need to compare each character in $r$ with a character in the candidate subsequence once. Since we have $|L(s)|$ candidate positions for $s$, the cost for verifying candidates is estimated as $\mu \times |r| \times |L(s)|$, where $\mu$ is a constant.[2] Therefore, the mapping cost of a signature $s$, denoted by $C(s)$, is estimated as

$$C(s) = \sum_{g \in Q} |I(g)| + \mu \times |r| \times |L(s)|. \quad (1)$$

In the following recurrence, $M[r, k+1]$ stores a minimum cost for mapping a read $r$ (i.e., $r[0 \,..\, |r|-1]$) when a maximum number of allowed errors is $k$, or the number of signatures to be selected is $k+1$.

$$M(r, k+1) = \texttt{min}\{C(r[i \,..\, j]) + M(r[j+1 \,..\, |r|-1], k)\},$$

where $0 \le i \le j - q + 1 \le |r| - (k+1) * q$, and $M(*, 0) = 0$. Algorithm 1 is a dynamic programming algorithm that selects an optimized combination of signatures from a read based on the recurrence above. The algorithm consists of the following two parts.

**Minimum-cost calculation** (the for loop in line 4): $M[i][n]$ in Algorithm 1 stores the minimum cost of $n$ signatures from $r[i \,..\, |r|-1]$. It corresponds to $M(r[i \,..\, |r|-1], n)$ in the recurrence above. Given a read $r$ with a hamming distance threshold $k$, the algorithm calculates the matrix $M$ to obtain $M[0][k+1]$, which is the minimum cost of $k+1$ signatures selected from $r[0 \,..\, |r|-1]$. The for loop calculates the matrix for the $n^{th}$ signature after calculating the matrix for $n-1$ signatures. It is worth noting that the algorithm as well as the recurrence above calculates costs of signatures starting from a suffix of the read. Hence, to select the $n^{th}$ signature, the algorithm needs to select $n-1$ signatures in a suffix of the read and $n\_sigs - n$ signatures in a prefix of the read. This gives a lower bound of the start position and an upper bound of the end position of the $n^{th}$ signature as depicted in Figure 4. Since the minimum length of a signature is $q$, which is the length of indexed grams, the lower bound of the start position $lb$ is $(n\_sigs - n) * q$ (line 5), and the upper bound of the end position $ub$ is $|r| - (n-1) * q - 1$ (line 6). Note that the upper bound of the start position is $ub - q + 1$.

[2]If we use a banded semi-global alignment algorithm for an edit distance threshold $k$, the cost for verifying a candidate is proportional to $k \times (|r| + k)$, and thus the verification cost in $C(s)$ is modified as $\mu \times k \times (|r| + k) \times |L(s)|$.

**Algorithm 1:** SignatureSelection($r$, $k$)

**input** : $r$ is a read,
$k$ is a maximum number of allowed errors
**output**: a set of $L(s)$'s for each selected signature $s$

```
1  n_sigs ← k + 1;
2  for i ← 0 to |r| − 1 do
3      M[i][0] ← 0;
4  for n ← 1 to n_sigs do
5      lb ← (n_sigs − n) ∗ q; // q is a gram length
6      ub ← |r| − (n − 1) ∗ q − 1;
7      for i ← lb to ub − q + 1 do
8          min ← ∞;
9          for j ← i + q − 1 to ub do
10             cost ← C(r[i .. j]) + M[j + 1][n − 1];
11             if cost < min then
12                 min ← cost; end ← j;
13         M[i][n] ← min;
14         E[i][n] ← end;
15     B[ub − q + 1][n] ← ub − q + 1;
16     for i ← ub − q down to lb do
17         if M[i][n] > M[i + 1][n] then
18             M[i][n] ← M[i + 1][n];
19             B[i][n] ← B[i + 1][n];
20         else
21             B[i][n] ← i;
22 sig ← ∅; begin ← 0;
23 for n ← n_sigs down to 1 do
24     begin ← B[begin][n];
25     end ← E[begin][n];
26     sig ← sig ∪ {r[begin .. end]};
27     begin ← end + 1;
28 return sig;
```

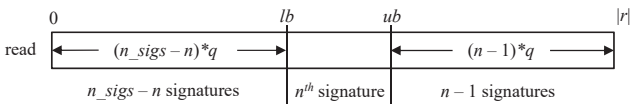


Fig. 4. Lower and upper bounds of the $n^{th}$ signature

For each start position $i$ of the $n^{th}$ signature (the for loop in line 7), the algorithm calculates costs of $n$ signatures (line 10) considering all possible end positions $j$ (the for loop in line 9). Then, it saves the minimum cost in $M[i][n]$ (line 13) with the corresponding end position in $E[i][n]$ (line 14). To make $M[i][n]$ keep the minimum cost of $n$ signatures from $r[i \,..\, |r|-1]$, the algorithm modifies $M[i][n]$ as follows (lines 15–21). For each start position $i$, it finds a position $b$ such that $M[b][n] = \texttt{min}_{s=i}^{ub-q+1} M[s][n]$, and replaces $M[i][n]$ with $M[b][n]$ (line 18). It also stores $b$ in $B[i][n]$ to immediately find the start position of the $n^{th}$ signature in $r[i \,..\, |r|-1]$ (in lines 15, 19 and 21).

**Signature generation** (the for loop in line 23): In opposite to the cost calculation, the algorithm selects an optimal combination of signatures from the beginning of the read. The start

position of the first signature is stored in $B[0][k+1]$ (line 24) and its end position is stored in $E[B[0][k+1]][k+1]$ (line 25). The algorithm selects remaining $k$ signatures in the same way (the for loop in line 23) and returns the selected signatures (line 28).

*Example 8:* Consider a read $r$ in Figure 3 assuming a 4-gram inverted index. Given a hamming distance threshold 2, Algorithm 1 calculates costs of three signatures as follows. To select the first signature, it leaves room for two remaining signatures in a prefix of the read. Since the minimum size of a signature is 4, the minimum prefix for the two remaining signatures is $r[0 .. 7]$, and the lower bound of the start position of the first signature is 8. It is obvious that the upper bound of the end position of the first signature is $|r|-1 = 14$. For a start position 8, the algorithm considers end positions of 11, 12, 13, and 14. In this example, we set $\mu$ to 1 in Equation 1. Hence, mapping costs of $r[8 .. 11]$, $r[8 .. 12]$, $r[8 .. 13]$, and $r[8 .. 14]$ are calculated as $42+15\times42 = 672$, $(42+26)+15\times15 = 293$, $(42 + 22) + 15 \times 7 = 169$, and $(42 + 20) + 15 \times 6 = 152$, respectively. Since $r[8 .. 14]$ has a minimum cost, the algorithm fills $M[8][1]$ with the cost 152 and saves the end position 14 in $E[8][1]$. It fills $M[*][1]$ and $E[*][1]$ for start positions of 9, 10, and 11 of the first signature in the same way. For the start position 8, the algorithm modifies the value of $M[8][1]$ to the value of $M[9][1] = 151$, which is the minimum cost among $M[i][1]$ where $8 \leq i \leq 11$. Then, it saves the start position 9 in $B[8][1]$. In the same way, it modifies $M[*][1]$ and fills $B[*][1]$ as depicted in Figure 5 ($M$/$E$/$B[i][j]$ corresponds to the $i^{th}$ column and the $j^{th}$ row of the matrix $M$/$E$/$B$ in the figure). Now, we consider the second signature. Its lower and upper bounds are 4 and 10, respectively. For the start position 4, the algorithm considers end positions of 7, 8, 9, and 10, and calculates mapping costs of $r[4 .. 7]$, $r[4 .. 8]$, $r[4 .. 9]$, and $r[4 .. 10]$. It adds $M[8][1]$, $M[9][1]$, $M[10][1]$, and $M[11][1]$ into the costs of $r[4 .. 7]$, $r[4 .. 8]$, $r[4 .. 9]$, and $r[4 .. 10]$, respectively. Then, it chooses a minimum cost, $C(r[4 .. 10])+M[11][1]$, and saves the cost in $M[4][2]$ and the end position of r[4 .. 10] in $E[4][2]$. $M[*][2]$ and $E[*][2]$ are filled in this way. The algorithm modifies $M[*][2]$ in the same way as $M[*][1]$ and saves the start positions in $B[*][2]$. It also processes segments for the third signature in the same way and fills the matrices as depicted in Figure 5. Once the matrices are filled, the algorithm selects an optimal combination of three signatures (star-shaped in Figure 3) as follows. It selects signatures starting from the beginning of the read. The start position $b_0 = 0$ of the first signature is recorded in $B[0][3]$. It finds the end position $e_0 = 5$ of the first signature in $E[b_0][3]$. Thus, the first signature is $r[0 .. 5]$. The start position $b_1 = 6$ of the second signature is stored in $B[e_0 + 1][2] = 6$. The end position of the second signature is $E[b_1][2] = 10$, and thus the second signature is $r[6 .. 10]$. Similarly, the algorithm identifies the third signature $r[11 .. 14]$.

| | 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9 | 10 | 11 | 12 | 13 | 14 |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|
| M[*][0] | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 |
| M[*][1] | - | - | - | - | - | - | - | - | 151 / 152 | 151 | 252 | 320 | - | - | - |
| M[*][2] | - | - | - | - | 425 | 479 / 496 | 479 | 800 | - | - | - | - | - | - | - |
| M[*][3] | 690 | 872 | 959 | 1248 | - | - | - | - | - | - | - | - | - | - | - |
| E[*][1] | - | - | - | - | - | - | - | - | 14 | 14 | 14 | 14 | - | - | - |
| E[*][2] | - | - | - | - | 9 | 10 | 10 | 10 | - | - | - | - | - | - | - |
| E[*][3] | 5 | 5 | 5 | 6 | - | - | - | - | - | - | - | - | - | - | - |
| B[*][1] | - | - | - | - | - | - | - | - | 9 | 9 | 10 | 11 | - | - | - |
| B[*][2] | - | - | - | - | 4 | 6 | 6 | 7 | - | - | - | - | - | - | - |
| B[*][3] | 0 | 1 | 2 | 3 | - | - | - | - | - | - | - | - | - | - | - |

Fig. 5. A running example of signature selection for a read in Figure 3

In our example, we assume that frequencies of segments of different lengths are given. Since we use a $q$-gram inverted index, however, frequencies of segments whose lengths are greater than $q$ are unknown. Hence, we need to estimate frequencies of those segments to select optimal signatures. Although we can use existing intersection-size-estimation techniques (e.g., [23], [24]) with simple modifications, they increase index building time as well as index size. In this paper, we estimate the frequency of a segment $s$ of a read as follows. We find a minimum frequency among the frequencies of $q$-grams in $G(s, q)$ using a $q$-gram inverted index on a reference sequence. It is obvious this frequency is an upper bound of the frequency of $s$, and we can use this upper bound as an estimated frequency value of $s$. This approach, however, makes multiple segments sharing a low-frequency $q$-gram have the same estimated frequency. To alleviate the problem, we make a simplified assumption that the length of a segment is inversely proportional to the frequency of the segment. Based on the assumption, we divide the upper bound by the difference between the length of $s$ and the $q$-gram length. Hence, the estimated frequency of $s$, denoted by $|\hat{L(s)}|$, is

$$|\hat{L(s)}| = \frac{\mathtt{min}_{g \in G(s,q)}\{|I(g)|\}}{|s| - q},$$

where $|s| > q$. Note that when $|s| = q$, $s$ contains one $q$-gram and we directly find $|L(s)|$ from a $q$-gram inverted index.

Hobbes2 [2] proposed to select $k + 2$ non-overlapping $q$-grams for generating candidate positions. It showed that adding an additional $q$-gram significantly reduces the number of candidates and improves the performance of read mapping. Based on the observation of Hobbes2, we take the approach of selecting $k + 2$ signatures in this paper. According to Hobbes2, it is still reasonable to assume that each signature independently generates candidate positions when we use $k+2$ signatures. Hence, we can select an optimal combination of $k + 2$ signatures by assigning $k + 2$ to $n\_sigs$ (line 1 of Algorithm 1).

## IV. Candidate Generation and Verification

In this section, we present candidate generation and verification techniques considering indel errors. We first define a *matched signature* of a candidate position as follows.

*Definition 4:* Given a signature $r[b .. e]$ of a read $r$ and a position $l$ of a reference sequence $\Gamma$, $r[b .. e]$ is a *matched signature* of $l$ if and only if $\Gamma[l + b .. l + e] = r[b .. e]$.

*Example 9:* Consider a reference sequence $\Gamma_1$ and a read $r$ in Figure 6. $r[3 .. 7]$ is a matched signature of position 1 of $\Gamma_1$ since $\Gamma_1[1 + 3 .. 1 + 7] = r[3 .. 7]$.

Unlike mismatch errors, indel errors introduce the following two problems in generating and verifying candidate positions.

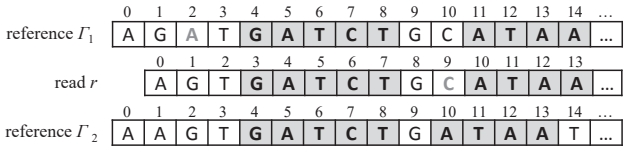


Fig. 6. Problems caused by indel errors

- A candidate position is determined by the relative position of a matched signature in a read as we described in Section II. If we consider indel errors, however, there is a problem that a candidate position cannot be exactly determined due to indel errors occurring before any matched signatures. Indel errors occurring after any matched signatures cause a similar problem that the length of a candidate subsequence is not deterministic.
- Because we use $k+2$ signatures for an edit distance threshold $k$, a candidate position of a reference needs to have at least two matched signatures of a read. Indel errors occurring between two signatures of a read can prevent a true mapping position from being a candidate position.

*Example 10:* Given a read $r$ and a reference sequence $\Gamma_1$ in Figure 6, we have two matched signatures of GATCT and ATAA as shown in the figure. According to the relative positions of matched signatures, position 1 of $\Gamma_1$ becomes a candidate position. However, a true candidate position is position 0 due to a deletion of A at position 2 of $\Gamma_1$. Let us consider another reference sequence $\Gamma_2$ in Figure 6. Position 0 and position 1 of $\Gamma_2$ have one matched signature, respectively. A candidate position needs to have at least two matched signatures in the $(k+2)$-signature scheme, and thus these positions will be dropped. If we take it into consideration to insert C between positions 9 and 10 of $\Gamma_2$, however, position 1 of $\Gamma_2$ becomes a candidate position.

Suppose we map a read $r$ against a reference sequence $\Gamma$ with a maximum number of allowed indel errors $k$. Given a candidate position $l$ of $\Gamma$, we need to consider up to $k$ positions left/right to $l$ to handle deletions/insertions occurring before any matched signatures. That is, every position in $[l-k, l+k]$ needs to be considered as a candidate position. Similarly, to handle indel errors occurring after any matched signatures, we need to consider the length of a candidate subsequence up to $|r|+k$ for the candidate position $l$. That is, the upper bound of the end position for the candidate position $l$ is $l+|r|+k-1$. Note that if a true mapping position is $l+\alpha$, where $\alpha \leq k$, the maximum number of allowed errors after any matched signatures is $k-\alpha$, since $\alpha$ deletions in $\Gamma$ occur before any matched signatures. Hence, the upper bound of the end position for the candidate $l$ is still $l+|r|+k-1$. Therefore, we can solve the problem of indel errors before/after any matched signatures by using a verification window of $\Gamma[l-k \,..\, l+|r|+k-1]$, which we call a *full-verification window*.

To reduce verification overheads, Hobbes and Hobbes2 do not consider a full verification window and they miss some true mapping positions. In Hobbes3, we perform a banded semi-global alignment implemented using Myers' bit-vector algorithm [15] on a full-verification window. In case that multiple mapping positions are found in a verification window, we choose one position with a minimum edit error as a mapping position.

To solve the problem caused by indel errors between two signatures of a read, we need to consider certain positions having only one matched signature as candidate positions as we illustrated in Example 10. Given two positions $l_1$ and $l_2$, each of which has only one matched signature and $l_1 - l_2 \leq k$, Hobbes2 makes both positions as candidate positions. In Hobbes3, however, we use either $l_1$ or $l_2$ as a candidate position based on the following lemma.

*Lemma 2:* Given a read $r$ and a candidate position $l$ of a reference sequence $\Gamma$, suppose we use a full-verification window of $\Gamma[l-k \,..\, l+|r|+k-1]$ for an edit distance threshold $k$. Consider two positions $l_1$ and $l_2$ in $\Gamma$ such that each of them has only one matched signature. If $|l_1 - l_2| \leq k$, it does not lose any true mapping position to make either $l_1$ or $l_2$ as a candidate position.

*Proof:* Let $s_1$ and $s_2$ be the matched signatures of $l_1$ and $l_2$, respectively, where the position of $s_1$ in $r$ is less than that of $s_2$. Here, we prove the case of $l_1 < l_2$, and the case of $l_1 > l_2$ can be proved similarly. As $l_1 < l_2$, we need $l_2 - l_1$ deletions between $s_1$ and $s_2$ in $\Gamma$ to map the read $r$. Hence, if there is no indel error before $s_1$, the start position of $r$ should be aligned with $l_1$.

Suppose we use $l_1$ as a candidate position. Indel errors before $s_1$ will be covered by the verification window. In this case, the end position of $r$ will be aligned with $l_1+|r|+(l_2-l_1)-1 = |r|+l_2-1$ of $\Gamma$ if there are no indel errors after $s_2$. Since we already have $l_2 - l_1$ errors before $s_2$, the maximum number of allowed indel errors after $s_2$ is $k-(l_2-l_1)$. Hence, the largest end position of $\Gamma$ is $(|r|+l_2-1)+\{k-(l_2-l_1)\} = l_1+|r|+k-1$, which is covered by the verification window. Therefore, we can use $l_1$ as a candidate position.

If we use $l_2$ as a candidate position instead of $l_1$, the largest end position of $\Gamma$ will be covered by the verification window since $l_1 < l_2$. The maximum number of allowed indel errors before $s_1$ is $k-(l_2-l_1)$ because $l_2-l_1$ deletion errors occur after $s_1$. The smallest start position of $\Gamma$ that can be aligned with the read is $l_1 - \{k-(l_2-l_1)\} = l_2 - k$, which is covered by the verification window. Therefore, we can use $l_2$ as a candidate position instead of $l_1$. ■

Algorithm 2 generates candidate positions using selected signatures of a read. For each signature $s$, a list of candidate positions, $L(s)$, is assumed to be built using a $q$-gram inverted index while selecting the signature, where positions in $L(s)$ are sorted in the ascending order of their values. The algorithm first copies candidate positions of the first signature into the candidate list (line 3). It then merges candidate positions of each remaining signature into the candidate list (for loop in line 5). To minimize costs of scanning lists, it sorts lists by their sizes before merging them (line 1). It finally returns those positions that appear at least twice (line 28 and line 29). For a position $l$ appearing only once, if there exists a position $l'$ such that $|l-l'| \leq k$, the algorithm makes $l$ as a candidate position according to Lemma 2 (else if statement in line 21 and else statement in line 24).

**Algorithm 2:** GenerateCandidates($k$, $s$)

**input** : $k$ is an edit distance threshold,
$s = \{s_0, \ldots, s_{k+1}\}$ is a set of $k+2$ signatures
**output**: a list of candidate positions $c$.

```
1  sort signatures in s by their frequencies;
2  c ← empty list of (pos, cnt) pairs;
3  foreach pos ∈ L(s_0) do
4      c.push_back(pos, 1);
5  for i ← 1 to k + 1 do
6      c' ← empty list of (pos, cnt) pairs;
7      j_1 ← 0 // index of the 1st element in c;
8      j_2 ← 0 // index of the 1st element in L(s_i);
       /* assume each list contains max value at the end */
9      while j_1 < c.size() or j_2 < L(s_i).size() do
10         if c[j_1].pos − L(s_i)[j_2] > k then
11             c'.push_back(L(s_i)[j_2], 1);
12             j_2 ← j_2 + 1;
13         else if c[j_1].pos − L(s_i)[j_2] < −k then
14             c'.push_back(c[j_1].pos, c[j_1].cnt);
15             j_1 ← j_1 + 1;
16         else
17             if c[j_1].pos = L(s_i)[j_2] then
18                 c'.push_back(c[j_1].pos, c[j_1].cnt + 1);
19                 j_1 ← j_1 + 1;
20                 j_2 ← j_2 + 1;
21             else if c[j_1].pos > L(s_i)[j_2] then
22                 c'.push_back(L(s_i)[j_2], 2);
23                 j_2 ← j_2 + 1;
24             else
25                 c'.push_back(c[j_1].pos, c[j_1].cnt + 1);
26                 j_1 ← j_1 + 1;
27     c ← c';
28 remove every pos from c whose cnt is less than 2;
29 return c;
```

## V. Experimental Results

In this section, we present experimental results of mapping single-end reads and paired-end reads, where a single-end read (also called just a read) is a subsequence of a reference genome sequence while a paired-end read is a pair of single-end reads. To map a paired-end read, we first map each single-end read in the pair and then identify if the difference between the mapping positions of the two single-end reads is within a predefined range.

### A. Experimental Setup

We implemented Hobbes3 in C++, and compiled it with GCC 4.4.3. All experiments were run on a machine with 32 GB of RAM, and Intel core i7 (4 cores and 8 threads total) at 3.4 GHz, running a 64-bit Ubuntu OS. We thoroughly compared Hobbes3 with five state-of-the-art so-called "*all-mappers*," which are designed to return all mapping positions of a read - Hobbes2, Bitmapper, Masai, Yara [25], and RazerS3, and three other popular read mappers - Gem [19], BWA, and Bowtie2. We did not include other all-mappers (such as SOAP2 [26], SHRiMP2 [27], mrsFAST [28], and mrFAST-CO [29]) in our comparison as it has been shown previously that these all-mappers do not perform as well as Hobbes2, Bitmapper, RazerS3, and Masai.

In the experiments, we used the human genome HG18 as the reference sequence. The human genome has many applications, and thus existing read mappers mainly focus on mapping reads against this sequence. Although previous work had additional experiments on smaller genomes, mapping reads against those smaller genomes is usually fast and less challenging than against the human genome. Due to limited space, we only focused on the human genome in our experiments. We generated simulated reads of length 100 bp and 500 bp, respectively, from HG18 using a read simulator, Mason [30], which was configured as the default profile setting with the `illumina` option. We also used real reads of length 100 bp from specimen HG00096 of the "1000 genome project" [31]. We mapped reads against the reference sequence using edit distance constraints. For the down-stream applications using all-mappers, the edit distance threshold is usually set to 4% or 5% of the read length [14]. We also set edit distance thresholds to around 5% errors of the read length in our experiments.

The weight of scanning cost and that of verification cost differ from each other, and the verification of a candidate is terminated early as soon as we find that the candidate does not meet the threshold. In Hobbes3, the parameter $\mu$ in Equation 1 was used to reflect these factors. Through experiments, we observed that the parameter was not very sensitive and we tuned it to 0.1.

### B. Configurations of Read Mappers

We configured read mappers to return all mapping positions and output results in the SAM format [32] with cigar strings. The following are the details of the configuration of each tool. Unless otherwise stated, other options of the mappers were configured as their default settings.

- **Hobbes**: we used Hobbes version 3.0 for Hobbes3 and version 2.1 for Hobbes2. In order to use Hobbes as an all-mapper, we specified the `-a` option. We used the `--indel` option for edit distance constraints. For the paired-end mapping, we set `--min` and `--max` parameters to 110 and 290, respectively. We used an 11-gram inverted index for edit distance 5 and 6, and a 10-gram inverted index for edit distance 7.
- **RazerS3**: version 3.4 was used. To use RazerS3 as an all-mapper, we set the parameter `-m` to 1,000,000. We set the parameters `-ll` and `-le` to 200 and 90, respectively, for the paired-end mapping.
- **Masai**: version 0.7.2 was used. To use Masai as an all-mapper, we used the '`-mm all`' option. We set `-ll` and `-le` parameters to 200 and 90, respectively, for the paired-end mapping.
- **Yara**: version 0.9.3 was used. We used the '`--all`' option to configure Yara as an all-mapper.
- **Bitmapper**: version 1.0.0.7 was used. Bitmapper is an all-mapper supporting edit distance only, and thus

we do not need to configure it. For the paired-end mapping, we set `--min` and `--max` parameters to 110 and 290, respectively. We built an index using 11-grams for the experiments.

- **BWA**: version 0.7.12 was used. We used the `-N` option to configure BWA as an all-mapper. To specify an edit distance 5, we set `-n`, `-o`, and `-e` parameters to 5, 5, and -1, respectively. We used the `aln` command to align reads and the `samse` command to convert the output into SAM format output. The output conversion time was included in the mapping time.
- **Bowtie2**: version 2.2.6 was used. We used the `--all` option to configure Bowtie2 as an all-mapper. To specify an edit distance 5, we used the `--end-to-end` and `--ignore-quals` options and set the `--mp`, `--np`, `--rdg`, `--rfg`, and `--score-min` parameters to [1,1], 1, [0,1], [0,1], and [L,0,−0.05], respectively.
- **Gem**: version 1.376 was used. To specify an edit distance 5, we used '`quality-format = ignore`' and set both `-m` and `-e` parameters to 5. To configure Gem as an all-mapper, we set `-d` parameter to `all` and `-s` parameter to 1,000,000. We converted the output of `gem-mapper` into SAM format output using `gem-2-sam`. We included the conversion time in the mapping time.

### C. Effects of Gram Length and Number of Signatures

A gram length of an inverted index greatly affects the performance of read mapping. By using a large $q$ value, we can shorten lengths of inverted lists, and thus reduce scanning cost for inverted lists as well as the number of candidates. In the $(k+2)$-signature scheme, the upper bound of $q$ is determined by an edit distance threshold $k$ and the length of a read $r$ as follows:

$$q_{upper} = \left\lfloor \frac{|r|}{k+2} \right\rfloor.$$

If we use $q_{upper}$, however, we lose opportunities to select good signatures. For 100-bp reads with $k = 7$, for instance, we have no chance to select a good combination of signatures, since there is only one combination of $k+2$ signatures for each read. To solve the problem, we use the following gram length:

$$q_{good} = \left\lfloor \frac{|r|}{k+3} \right\rfloor.$$

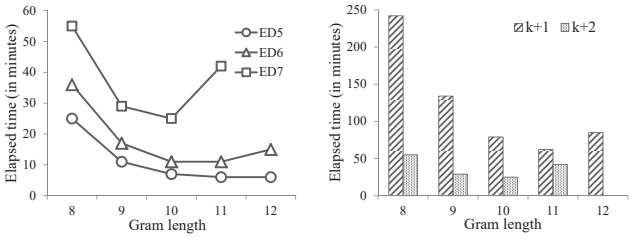


Fig. 7. Effects of gram length and number of signatures. Note that the maximal allowable gram length is 11 for the $(k+2)$-signature scheme when ED=7.

We ran an experiment to evaluate the effect of a gram length $q$ on the performance. In the experiment, we mapped 1 million 100-bp real reads against HG18 using 8 threads for various edit distance thresholds and gram lengths. Figure 7(a) shows the results. As shown in the figure, we obtained the best performance with $q_{good}$ for each edit distance threshold.

We also compared the $(k+1)$-signature scheme with the $(k+2)$-signature scheme. For this experiment, we used edit distance threshold 7. Figure 7(b) shows the results. The observation of Hobbes2 is also valid for the proposed variable-length signatures, and the $(k+2)$-signature scheme outperformed the $(k+1)$-signature scheme in all gram lengths as shown in the figure.

### D. Index Construction and Memory Footprint

We built an inverted index of overlapping $q$-grams on the reference genome HG18. Hobbes3 eliminates bit vectors from an index, and thus the size of an index on disk is the same as that in memory. For the human genome HG18, the index size is about 11 GB. Hobbes2 uses 16-bit vectors, resulting in an index size of 17 GB for HG18. Although Hobbes2 keeps about 11GB index in memory by removing bit vectors while loading index into memory, it needs to scan the whole index on disk to load the index. Hence, the index loading step of Hobbes2 is slower than that of Hobbes3. For HG18, Hobbes3 loads an index about two times faster than Hobbes2. Since Hobbes3, similar to Hobbes2, has a tight-knit multi-threaded framework that parallelizes both indexing and mapping, it took only a few minutes to build an index for HG18.

### E. Single-end Mapping

We used the Rabema [33] benchmark to compare accuracy of read mappers. In Rabema, the accuracy is defined as follows. Each read gives at most one point. If a read matches at $n$ positions, each found position gives $1/n$ point. To get percentages, the number of achieved points is divided by the number of reads and multiplied by 100. Rabema categorizes mapping accuracy into all, all-best, and any-best. All denotes all mappings within a given edit distance threshold, all-best denotes all mappings with the lowest edit distance, and any-best denotes any mapping with the lowest edit distance. The following example demonstrates the accuracy and mapping categories.

*Example 11:* Given a reference sequence and three reads with an edit distance threshold 2, consider the numbers of all possible mapping positions of each read and mapping results of a read mapper in the following table:

| read | # of all possible mapping positions | | | | results of a mapper | | | |
|---|---|---|---|---|---|---|---|---|
| | ED2 | ED1 | ED0 | total | ED2 | ED1 | ED0 | total |
| $r_1$ | 12 | 8 | 0 | 20 | 11 | 7 | 0 | 18 |
| $r_2$ | 12 | 6 | 4 | 22 | 10 | 6 | 3 | 19 |
| $r_3$ | 15 | 8 | 0 | 23 | 12 | 7 | 0 | 19 |

In the all category, the mapper has a $86.324\%$ accuracy since $r_1$, $r_2$, and $r_3$ achieve 18/20, 19/22, and 19/23 point, respectively, and the accuracy is computed as $(18/20+19/22+19/23)/3 \times 100\% = 86.324\%$. We also compute the accuracy with each distinct edit distance. For instance, the mapper has a $91.6666\%$ accuracy for those mappings having an edit distance 1 as $(7/8 + 6/6 + 7/8)/3 \times 100\% = 91.6666\%$. In

TABLE I. RABEMA BENCHMARK RESULTS OF MAPPING 100K SIMULATED READS OF LENGTH 100 BP AGAINST HG18 (ED 5)

| read mapper | time (min:sec) 1 thr | 8 thrs | accuracy ( total ED0 ED1 ED2 ED3 ED4 ED5 ) all | | | | | all-best | | | | | any-best | | | | | recall | | | | peak memory |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|
| Hobbes3 | 3:01 | 1:49 | 99.9999 | 100.0 | 100.0 | 100.0 | | 100.00 | 100.0 | 100.0 | 100.0 | | 100.00 | 100.0 | 100.0 | 100.0 | | 99.092 | 100.0 | 100.0 | 100.0 | | 14.6 GB |
| | | | | 100.0 | 100.0 | 100.0 | | | 100.0 | 100.0 | 100.0 | | | 100.0 | 100.0 | 100.0 | | | 100.0 | 100.0 | 100.0 | |
| Hobbes2 | 8:46 | 4:06 | 99.8573 | 100.0 | 100.0 | 100.0 | | 99.9979 | 100.0 | 100.0 | 100.0 | | 99.998 | 100.0 | 100.0 | 100.0 | | 98.971 | 100.0 | 99.90 | 99.68 | | 14.7 GB |
| | | | | 99.99 | 99.94 | 97.48 | | | 100.0 | 100.0 | 99.84 | | | 100.0 | 100.0 | 99.84 | | | 99.34 | 99.04 | 99.77 | |
| Bitmapper | 3:23 | 2:05 | 99.9994 | 100.0 | 100.0 | 100.0 | | 100.00 | 100.0 | 100.0 | 100.0 | | 100.00 | 100.0 | 100.0 | 100.0 | | 99.092 | 100.0 | 100.0 | 100.0 | | 15.1 GB |
| | | | | 100.0 | 99.99 | 99.99 | | | 100.0 | 100.0 | 100.0 | | | 100.0 | 100.0 | 100.0 | | | 100.0 | 100.0 | 100.0 | |
| Masai | 9:56 | – | 99.8337 | 100.0 | 100.0 | 100.0 | | 99.9455 | 100.0 | 100.0 | 100.0 | | 99.9455 | 100.0 | 100.0 | 100.0 | | 99.038 | 100.0 | 100.0 | 100.0 | | 16.9 GB |
| | | | | 99.73 | 99.18 | 97.69 | | | 99.69 | 98.73 | 98.52 | | | 99.69 | 98.73 | 98.52 | | | 99.71 | 98.77 | 98.56 | |
| Yara | 2:04 | 1:00 | 91.6214 | 98.87 | 97.90 | 94.46 | | 97.6622 | 97.65 | 97.82 | 97.68 | | 99.9354 | 100.0 | 100.0 | 99.87 | | 96.693 | 97.69 | 97.73 | 97.32 | | 5.3 GB |
| | | | | 81.95 | 55.03 | 22.57 | | | 97.32 | 96.46 | 96.54 | | | 99.54 | 99.06 | 99.07 | | | 96.85 | 96.42 | 96.44 | |
| RazerS3 | 18:58 | 13:06 | 99.9089 | 100.0 | 100.0 | 100.0 | | 99.999 | 100.0 | 100.0 | 100.0 | | 99.999 | 100.0 | 100.0 | 100.0 | | 99.091 | 100.0 | 100.0 | 100.0 | | 4.6 GB |
| | | | | 100.0 | 99.86 | 98.44 | | | 100.0 | 100.0 | 99.92 | | | 100.0 | 100.0 | 99.92 | | | 100.0 | 100.0 | 99.92 | |
| Bowtie2 | – | 296:31 | 99.7493 | 100.0 | 100.0 | 100.0 | | 99.973 | 100.0 | 100.0 | 100.0 | | 99.9748 | 100.0 | 100.0 | 100.0 | | 98.85 | 100.0 | 99.71 | 99.46 | | 22.4 GB |
| | | | | 100.0 | 99.55 | 95.75 | | | 100.0 | 99.70 | 98.35 | | | 100.0 | 99.72 | 98.45 | | | 99.17 | 98.77 | 98.56 | |
| BWA | 41:41 | 11:35 | 97.7318 | 100.0 | 99.98 | 99.64 | | 98.899 | 100.0 | 99.98 | 99.61 | | 98.9032 | 100.0 | 99.98 | 99.61 | | 97.607 | 100.0 | 99.49 | 98.58 | | 7.4 GB |
| | | | | 93.47 | 82.91 | 75.15 | | | 93.03 | 78.87 | 70.57 | | | 93.03 | 78.98 | 70.73 | | | 91.59 | 78.28 | 71.18 | |
| Gem | 2:31 | 1:11 | 97.7401 | 100.0 | 99.99 | 99.84 | | 99.8657 | 100.0 | 99.88 | 99.81 | | 99.9294 | 100.0 | 99.96 | 99.93 | | 98.665 | 100.0 | 99.42 | 99.12 | | 7.0 GB |
| | | | | 97.36 | 88.78 | 68.31 | | | 99.47 | 99.28 | 97.34 | | | 99.69 | 99.61 | 97.67 | | | 98.17 | 98.29 | 98.64 | |
| Bowtie2* | 0:25 | 0:16 | 91.3477 | 98.87 | 97.75 | 93.55 | | 97.0844 | 97.65 | 97.32 | 95.69 | | 99.2916 | 100.0 | 99.45 | 97.65 | | 95.966 | 97.79 | 96.86 | 94.95 | | 3.3 GB |
| | | | | 81.07 | 53.90 | 21.95 | | | 95.38 | 93.98 | 93.74 | | | 97.41 | 96.24 | 95.89 | | | 94.67 | 93.33 | 93.49 | |
| BWA* | 0:55 | 0:46 | 92.2733 | 100.0 | 99.82 | 96.90 | | 98.793 | 100.0 | 99.83 | 99.41 | | 98.8386 | 100.0 | 99.89 | 99.49 | | 97.242 | 100.0 | 99.04 | 97.56 | | 4.5 GB |
| | | | | 79.11 | 45.49 | 16.99 | | | 92.57 | 78.26 | 70.34 | | | 92.70 | 78.60 | 70.73 | | | 90.19 | 77.11 | 70.35 | |
| Gem* | 0:14 | 0:06 | 94.3818 | 100.0 | 99.37 | 97.56 | | 99.8612 | 100.0 | 99.88 | 99.80 | | 99.9263 | 100.0 | 99.95 | 99.92 | | 98.613 | 100.0 | 99.28 | 99.04 | | 4.3 GB |
| | | | | 89.82 | 68.40 | 34.19 | | | 99.39 | 99.16 | 97.34 | | | 99.72 | 99.61 | 97.75 | | | 98.19 | 98.29 | 98.71 | |

TABLE II. RESULTS OF MAPPING 1 MILLION REAL READS OF LENGTH 100 BP AGAINST HG18 (ED 5)

| read mapper | time (min:sec) 1 thr | 8 thrs | mapped reads | accuracy ( total ED0 ED1 ED2 ED3 ED4 ED5 ) all | | | | all-best | | | | any-best | | | | peak memory |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|
| Hobbes3 | 21:07 | 5:43 | 915587 | 99.9997 | 100.0 | 100.0 | 100.0 | 99.9999 | 100.0 | 100.0 | 100.0 | 100.00 | 100.0 | 100.0 | 100.0 | 14.6 GB |
| | | | | | 100.0 | 100.0 | 99.99 | | 100.0 | 100.0 | 100.0 | | 100.0 | 100.0 | 100.0 | |
| Hobbes2 | 59:49 | 15:25 | 915587 | 99.9096 | 100.0 | 100.0 | 100.0 | 99.9989 | 100.0 | 100.0 | 100.0 | 100.00 | 100.0 | 100.0 | 100.0 | 14.9 GB |
| | | | | | 100.0 | 99.94 | 98.47 | | 100.0 | 99.99 | 99.93 | | 100.0 | 100.0 | 100.0 | |
| Bitmapper | 20:27 | 5:11 | 915587 | 99.9989 | 100.0 | 100.0 | 100.0 | 99.9998 | 100.0 | 100.0 | 100.0 | 100.00 | 100.0 | 100.0 | 100.0 | 15.4 GB |
| | | | | | 100.0 | 99.99 | 99.98 | | 100.0 | 100.0 | 99.99 | | 100.0 | 100.0 | 100.0 | |
| Masai | 46:02 | – | 915559 | 99.9372 | 100.0 | 100.0 | 100.0 | 99.9959 | 100.0 | 100.0 | 100.0 | 99.9968 | 100.0 | 100.0 | 100.0 | 17.3 GB |
| | | | | | 100.0 | 99.97 | 98.93 | | 100.0 | 99.98 | 99.71 | | 100.0 | 100.98 | 99.79 | |
| Yara | 176:54 | 80:10 | 915377 | 89.9918 | 98.82 | 94.90 | 83.00 | 96.7318 | 97.52 | 96.40 | 94.09 | 99.9766 | 100.0 | 100.0 | 100.0 | 5.3 GB |
| | | | | | 66.23 | 46.36 | 20.77 | | 92.30 | 90.74 | 86.98 | | 99.98 | 99.77 | 98.66 | |
| Gem | 30:28 | – | 914405 | 97.9387 | 100.0 | 99.99 | 99.84 | 99.8441 | 100.0 | 99.99 | 99.96 | 99.8662 | 100.0 | 100.0 | 99.99 | 10.3 GB |
| | | | | | 96.67 | 90.49 | 71.86 | | 99.74 | 99.55 | 90.13 | | 99.90 | 99.81 | 99.90.82 | |

TABLE III. RESULTS OF MAPPING 1 MILLION REAL READS OF LENGTH 100 BP AGAINST HG18 (ED 6)

| read mapper | time (min:sec) 1 thr | 8 thrs | mapped reads | accuracy ( total ED0 ED1 ED2 ED3 ED4 ED5 ED6 ) all | | | | | all-best | | | | | any-best | | | | | peak memory |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|
| Hobbes3 | 47:45 | 11:36 | 925878 | 99.9991 | 100.0 | 100.0 | 100.0 | 100.0 | 100.00 | 100.0 | 100.0 | 100.0 | 100.0 | 100.00 | 100.0 | 100.0 | 100.0 | 100.0 | 14.9 GB |
| | | | | | 100.0 | 99.99 | 99.99 | | | 100.0 | 100.0 | 100.0 | | | 100.0 | 100.0 | 100.0 | | |
| Hobbes2 | 184:08 | 41:47 | 925877 | 99.8807 | 100.0 | 100.0 | 100.0 | 100.0 | 99.9989 | 100.0 | 100.0 | 100.0 | 100.0 | 99.9998 | 100.0 | 100.0 | 100.0 | 100.0 | 15.1 GB |
| | | | | | 99.99 | 99.88 | 98.13 | | | 100.0 | 99.98 | 99.93 | | | 100.0 | 99.99 | 99.99 | | |
| Bitmapper | 38:00 | 8:57 | 892955 | 96.3954 | 95.62 | 96.22 | 97.82 | 98.83 | 96.391 | 96.00 | 96.53 | 97.89 | 98.72 | 96.4138 | 96.02 | 96.55 | 97.92 | 98.74 | 15.6 GB |
| | | | | | 99.92 | 99.54 | 99.78 | | | 98.95 | 99.06 | 99.10 | | | 98.98 | 99.12 | 99.20 | | |
| Masai | 101:01 | – | 925811 | 99.8648 | 100.0 | 100.0 | 100.0 | 100.0 | 99.9901 | 100.0 | 100.0 | 100.0 | 100.0 | 99.9924 | 100.0 | 100.0 | 100.0 | 100.0 | 17.3 GB |
| | | | | | 99.98 | 99.72 | 97.96 | | | 99.99 | 99.91 | 99.24 | | | 99.99 | 99.94 | 99.41 | | |
| Yara | 254:52 | 97:34 | 925773 | 88.2794 | 98.84 | 95.03 | 83.75 | 68.93 | 96.6238 | 97.52 | 96.41 | 94.09 | 92.33 | 99.9884 | 100.0 | 100.0 | 100.0 | 100.0 | 6.2 GB |
| | | | | | 54.36 | 34.99 | 15.29 | | | 90.95 | 88.28 | 84.99 | | | 99.99 | 99.99 | 98.98 | | |
| Gem | 60:08 | – | 924806 | 97.3451 | 100.0 | 99.99 | 99.88 | 99.36 | 99.8526 | 100.0 | 99.99 | 99.97 | 99.86 | 99.877 | 100.0 | 100.0 | 99.99 | 99.93 | 14.6 GB |
| | | | | | 94.31 | 86.54 | 66.53 | | | 99.62 | 99.24 | 88.97 | | | 99.82 | 99.68 | 89.85 | | |

TABLE IV. RESULTS OF MAPPING 1 MILLION REAL READS OF LENGTH 100 BP AGAINST HG18 (ED 7)

| read mapper | time (min:sec) 1 thr | 8 thrs | mapped reads | accuracy ( total ED0 ED1 ED2 ED3 ED4 ED5 ED6 ED7 ) all | | | | | all-best | | | | | any-best | | | | | peak memory |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|
| Hobbes3 | 119:43 | 25:03 | 934654 | 99.9983 | 100.0 | 100.0 | 100.0 | 100.0 | 100.00 | 100.0 | 100.0 | 100.0 | 100.0 | 100.00 | 100.0 | 100.0 | 100.0 | 100.0 | 14.9 GB |
| | | | | | 100.0 | 100.0 | 99.99 | 99.98 | | 100.0 | 100.0 | 100.0 | 100.0 | | 100.0 | 100.0 | 100.0 | 100.0 | |
| Hobbes2 | 491:23 | 107:30 | 934654 | 99.8612 | 100.0 | 100.0 | 100.0 | 100.0 | 99.9984 | 100.0 | 100.0 | 100.0 | 100.0 | 100.00 | 100.0 | 100.0 | 100.0 | 100.0 | 15.1 GB |
| | | | | | 100.0 | 99.99 | 99.87 | 97.97 | | 100.0 | 100.0 | 100.0 | 99.83 | | 100.0 | 100.0 | 100.0 | 100.0 | |
| Bitmapper | 83:10 | 26:45 | 934655 | 99.9888 | 100.0 | 100.0 | 100.0 | 100.0 | 99.9964 | 100.0 | 100.0 | 100.0 | 100.0 | 100.00 | 100.0 | 100.0 | 100.0 | 100.0 | 15.7 GB |
| | | | | | 100.0 | 99.97 | 99.91 | 99.88 | | 100.0 | 99.95 | 99.91 | 99.80 | | 100.0 | 100.0 | 100.0 | 100.0 | |
| Masai | 165:10 | – | 934586 | 99.8255 | 100.0 | 100.0 | 100.0 | 100.0 | 99.99 | 100.0 | 100.0 | 100.0 | 100.0 | 99.9925 | 100.0 | 100.0 | 100.0 | 100.0 | 17.3 GB |
| | | | | | 99.99 | 99.96 | 99.54 | 97.60 | | 99.99 | 99.98 | 99.84 | 99.16 | | 99.99 | 99.98 | 99.89 | 99.36 | |
| Yara | 233:50 | 97:33 | 934544 | 86.5594 | 98.86 | 95.13 | 84.17 | 70.28 | 96.505 | 97.52 | 96.41 | 94.09 | 92.33 | 99.998 | 100.0 | 100.0 | 100.0 | 100.0 | 6.2 GB |
| | | | | | 57.86 | 43.17 | 26.42 | 11.70 | | 90.97 | 88.29 | 85.97 | 82.75 | | 99.99 | 100.0 | 99.99 | 98.75 | |
| Gem | 167:41 | – | 933567 | 96.473 | 100.0 | 99.99 | 99.89 | 99.46 | 99.8376 | 100.0 | 99.99 | 99.97 | 99.87 | 99.8717 | 100.0 | 100.0 | 99.99 | 99.93 | 15.6 GB |
| | | | | | 95.55 | 90.86 | 80.99 | 60.58 | | 99.66 | 99.27 | 98.75 | 86.60 | | 99.85 | 99.71 | 99.37 | 88.06 | |

the all-best category, the mapper has a $83.3333\%$ accuracy, since the mapper achieves $7/8$, $3/4$, and $7/8$ points with the lowest edit distances of the reads, which results in an accuracy of $(7/8+3/4+7/8)/3\times 100\% = 83.3333\%$. The lowest edit distance of $r_1$ and $r_3$ is 1, thus the mapper has an accuracy of $(7/8+7/8)/2\times 100\% = 87.5\%$ with an edit distance 1 in the all-best category. In the any-best category, the mapper has a $100\%$ accuracy, since each read has at least one mapping for its lowest edit distance.

We used RazerS3 in its full-sensitive mode to build gold standards, which contain all possible mapping information for the simulated reads with an edit distance 5 and the real reads with edit distance 5, 6, and 7, respectively. For the performance comparison, we used RazerS3 with its default setting (not full-sensitive mode) as described in Section V-B.

Table I shows the mapping time and accuracy of read mappers for $100,000$ simulated reads with an edit distance

threshold 5. In the accuracy column, *total* denotes accuracy of total mappings within the threshold and *ED* $i$ denotes accuracy of those mappings with each distinct edit distance $i$. We ran the mappers using a single thread and $8$ threads, respectively, and measured the mapping time. We did not include the mapping time of Masai using 8 threads since it does not support multi-threading. We omitted the mapping time of Bowtie2 of a single thread since it could finish only about 10% of 100k reads in $6$ hours. As the simulator generated original positions of simulated reads, we also measured the recall of each mapper, which is the fraction of reads correctly reported at the original positions.

In terms of accuracy, the top three performers were Hobbes3, Bitmapper, and Hobbes2 with accuracy scores of $99.9999\%$, $99.9994\%$, and $99.8573\%$, respectively. Hobbes3 apparently improved Hobbes2 and returned almost all mapping positions of reads. Bitmapper also returned most mapping positions and was slightly worse than Hobbes3. For the recall, Hobbes3 and Bitmapper could return original positions of all reads within 5 edit errors, while Hobbes2 missed many original positions. RazerS3, Bowtie2, and Masai also showed good performance, but missed many mapping positions for high edit errors. BWA, Gem, and Yara missed too many mapping positions to be used as all-mappers. In terms of mapping time, Gem and Yara were the fastest but they showed poor accuracy. Among mappers exhibiting high accuracy, Hobbes3 ran the fastest, followed by Bitmapper.

We also ran Bowtie2, BWA, and Gem in their default mapping modes (or best mapping modes). We report the results at the end of Table I with mapper names Bowtie2*, BWA*, and Gem*. These mappers ran much faster than other mappers but exhibited very poor mapping results. In particular, they missed most mapping positions for large edit errors.

Tables II, III, and IV show experimental results on 1 million real reads with edit distance thresholds 5, 6, and 7, respectively. We included the total number of mapped reads in the third columns of the tables. Bowtie2, BWA, and RazerS3 were excluded from the experiments as they ran too slowly compared to the other all-mappers. We also excluded Bowtie2*, BWA*, and GEM* since they exhibited very poor accuracy as shown in Table I.

The accuracy of Hobbes3 was the best in all settings. Hobbes3 found almost all mapping positions and missed only a small fraction of mappings for high errors when multiple mappings were found in a verification window. Hobbes3 always returned mapping positions for every mappable read in all settings. In terms of mapping time, Hobbes3 and Bitmapper ran the fastest among the mappers - about three to four times faster than Hobbes2, about two to three times faster than Masai, up to sixteen times faster than Yara, and about 1.5 times faster than Gem. As we balance between the overhead of generating candidates and that of verifying candidates, it is not straightforward to explain the improvement of mapping speed of Hobbes3 in terms of the number of candidates only. That is, an optimal set of signatures do not minimize the number of candidates. Nonetheless, Hobbes3 tends to generate small number of candidate positions with a reasonable amount of time. For example, Hobbes3 using 8 threads took $436$ seconds to generate $984,316,998$ candidates for edit distance 7, while Hobbes 2 using $8$ threads took about $423$ seconds to generate $3,960,444,929$ candidates for the same edit distance.

Although Bitmapper ran slightly faster than Hobbes3 in most settings except that of 8 threads with edit distance 7, it showed poor accuracy than Hobbes3. In particular, Bitmapper generated unacceptably poor results with edit distance 6. Bitmapper also returned many incorrect mapping results that aligned reads across chromosome boundaries. With edit distance 7, for instance, the number of all mappable reads is $934,654$, but Bitmapper returned mapping positions for $934,655$ reads. Moreover, Hobbes3 is more scalable than Bitmapper since multi-threaded Hobbes3 ran faster for larger edit errors. For instance, it took about 56 minutes for Hobbes3 of 8 threads to map reads with edit distance threshold 8, while Bitmapper using 8 threads needed 64 minutes for mapping the reads. For the threshold, we used a 9-gram index of Hobbes3 while an 11-gram index of Bitmapper, because Bitmapper did not support gram lengths smaller than 11.

Masai exhibited good performance in terms of accuracy and time, but it does not support multi-threading. Yara, a successor of Masai, showed the worst performance. Although we configured Yara as an all-mapper with the `--all` option, the mapper missed most mapping positions for high edit errors. Gem, configured as an all-mapper, ran fast in the experiments on simulated reads, but ran much slower than other mappers such as Hobbes, Bitmapper, and Masai on real reads. Gem used most of the time and memory space in converting its mapping results into the SAM format. Using 8 threads, Gem was not able to convert its results into the SAM format, and was killed with an out-of-memory exception in our experimental setting.

We also ran an experiment on long reads. For 100,000 simulated reads of length 500 bp with 4% errors (or edit distance threshold 20), Hobbes3 with 8 threads returned mappings for all mappable reads in 5 minutes, while Bitmapper, Hobbes2, and Gem failed to map the reads. RazerS3 with 8 threads succeeded to map the reads, but it took about 20 minutes.

### F. Paired-end Mapping

We performed experiments for paired-end mapping with an edit distance 5 and the results are summarized in Table V. In the experiments, we compared five top all-mappers and excluded those mappers mainly designed to find a few top mapping positions. Since Masai does not directly support mapping paired-end reads, we separately ran `masai_mapper` for each read file to output results in the Masai's raw format, and merged the results using `masai_output_pe` to produce mappings in the SAM format. We included the conversion time in the mapping time of Masai.

TABLE V. RESULTS OF MAPPING 1 MILLION × 2 PAIRED-END READS OF LENGTH 100 BP AGAINST HG18 (ED 5)

| mapper | paired reads | mapping time (min:sec) 1 thr | 8 thrs | peak memory |
|---|---|---|---|---|
| Hobbes3 | 866887 | 21:48 | 06:06 | 14.8 GB |
| Hobbes2 | 866609 | 42:08 | 12:00 | 15.3 GB |
| Bitmapper | 866892 | 22:41 | 06:10 | 17.5 GB |
| Masai | 840798 | 49:42 | – | 17.3 GB |
| RazerS3 | 866802 | 91:59 | 66:35 | 17.7 GB |

We observed that Hobbes3 and Bitmapper were the fastest among all-mappers in both single threaded and multi-threaded cases. When using a single thread, Hobbes3 was about 2 times faster than Hobbes2 and Masai, and about 4.5 times faster than RazerS3. When using 16 threads, Hobbes3 was about 10 times faster than RazerS3. In terms of mapped pairs, Hobbes3 and Bitmappers were apparently better than other mappers. Although Bitmappers returned slightly more mapped pairs than Hobbes3, it included incorrectly aligned mapping positions that cross chromosome boundaries.

## VI. Conclusions

Hobbes3 is a read mapper designed to return all mapping positions of a read in a reference sequence. We have developed Hobbes3 using a technique that generates candidate mapping positions for a read using variable-length signatures. We showed that we could generate candidates for variable-length signatures using a fixed-length $q$-gram inverted index. Unlike previous techniques that focus on reducing the number of candidates only, the proposed technique considers a trade-off between candidate generation and verification overheads. We proposed a novel dynamic programming algorithm that balances the trade-off in an optimal way. Our experiments showed that Hobbes3 substantially improves the performance of read mapping in terms of speed and accuracy. Hobbes3 ran much faster than state-of-the-art all-mappers while returning almost all possible mapping positions.

## Acknowledgment

This research was supported by the MSIP (Ministry of Science, ICT and Future Planning), Korea, under the ITRC (Information Technology Research Center) support program (IITP-2015-R0992-15-1023) supervised by the IITP (Institute for Information & communications Technology Promotion). Chen Li was partially supported by the Joint Research Fund for Overseas Natural Science of China under grant 61129002. Xiaohui Xie was partially supported by NIH grant R0HG006870.

## References

[1] A. Ahmadi, A. Behm, N. Honnalli, C. Li, and X. Xie, "Hobbes: optimized gram-based methods for efficient read alignment," *Nucleic Acids Res.*, vol. 40, p. e41, 2012.

[2] J. Kim, C. Li, and X. Xie, "Improving read mapping using additional prefix grams," *BMC Bioinformatics*, vol. 15, p. 42, 2014.

[3] C. Li, B. Wang, and X. Yang, "VGRAM: Improving performance of approximate queries on string collections using variable-length grams," in *VLDB*, 2007, pp. 303–314.

[4] X. Yang, B. Wang, and C. Li, "Cost-based variable-length-gram selection for string collections to support approximate queries efficiently," in *SIGMOD Conference*, 2008, pp. 353–364.

[5] G. Li, D. Deng, J. Wang, and J. Feng, "Pass-join: A partition based method for similarity joins," *PVLDB*, vol. 5, pp. 253–264, 2011.

[6] J. Wang, G. Li, D. Deng, Y. Zhang, and J. Feng, "Two birds with one stone: An efficient hierarchical framework for top-k and threshold-based string similarity search," in *ICDE*, 2015, pp. 519–530.

[7] J. Kim, "An effective candidate generation method for improving performance of edit similarity query processing," *Information Systems*, vol. 41, pp. 116–128, 2015.

[8] R. Bayardo, Y. Ma, and R. Srikant, "Scaling up all-pairs similarity search," in *WWW Conference*, 2007, pp. 131–140.

[9] S. Sarawagi and A. Kirpal, "Efficient set joins on similarity predicates," in *SIGMOD Conference*, 2004, pp. 743–754.

[10] C. Xiao, W. Wang, and X. Lin, "Ed-join: an efficient algorithm for similarity joins with edit distance constraints," in *VLDB*, 2008, pp. 933–944.

[11] C. Li, J. Lu, and Y. Lu, "Efficient merging and filtering algorithms for approximate string searches," in *ICDE*, 2008, pp. 257–266.

[12] D. Weese, M. Holtgrewe, and K. Reinert, "Razers3: faster, fully sensitive read mapping," *Bioinformatics*, vol. 28, pp. 2592–2599, 2012.

[13] E. Siragusa, D. Weese, and K. Reinert, "Fast and accurate read mapping with approximate seeds and multiple backtracking," *Nucleic Acids Res.*, vol. 41, p. e78, 2013.

[14] H. Cheng, H. Jiang, J. Yang, Y. Xu1, and Y. Shang, "Bitmapper: an efficient all-mapper based on bit-vector computing," *BMC Bioinformatics*, vol. 16, p. 192, 2016.

[15] G. Myers, "A fast bit-vector algorithm for approximate string matching based on dynamic programming," *Journal of ACM*, vol. 46, pp. 395–415, 1999.

[16] B. Langmead, C. Trapnell, M. Pop, and S. Salzberg, "Ultrafast and memory-efficient alignment of short dna sequences to the human genome," *Genome Biol.*, vol. 10, p. r25, 2009.

[17] B. Langmead and S. Salzberg, "Fast gapped-read alignment with bowtie 2," *Nature Methods*, vol. 9, pp. 357–359, 2012.

[18] H. Li and R. Durbin, "Fast and accurate short read alignment with burrows-wheeler transform," *Bioinformatics*, vol. 25, pp. 1754–1760, 2009.

[19] S. Marco-Sola, M. Sammeth, R. Guigó, and P. Ribeca, "The gem mapper: fast, accurate and versatile alignment by filtration," *Nature Methods*, vol. 9, pp. 1185–8, 2012.

[20] D. Newkirk, J. Biesinger, A. Chon, K. Yokomori, and X. Xie, "Arem: aligning short reads from chip-sequencing by expectation maximization," *J. Comput. Biol.*, vol. 18, pp. 1495–1505, 2011.

[21] A. Roberts and L. Pachter, "Streaming fragment assignment for real-time analysis of sequencing experiments," *Nature Methods*, vol. 10, pp. 71–73, 2013.

[22] Y. Lil and X. Xie, "A mixture model for expression deconvolution from rna-seq in heterogeneous tissues," *BMC Bioinformatics*, vol. 14(Suppl 5):S11, 2013.

[23] A. Mazeika, M. H. Böhlen, and N. Koudas, "Estimating the selectivity of approximate string queries," *ACM Transaction on Database Systems*, vol. 32, p. 12, 2007.

[24] Z. Chen, F. Korn, N. Koudas, and S. Muthukrishnan, "Selectivity estimation for boolean queries," in *PODS*, 2000, pp. 216–225.

[25] E. Siragusa and K. Reinert. Yara: well-defined alignment of high-throughput sequencing reads. [Online]. Available: http://www.seqan.de/project/yara

[26] R. Li, C. Yu, Y. Li, T.-W. Lam, K. K. S.-M. Yiu, and J. Wang, "Soap2: an improved ultrafast tool for short read alignment," *Bioinformatics*, vol. 25, pp. 1966–1967, 2009.

[27] M. David, M. Dzamba, D. Lister, L. Ilie, and M. Brudno, "Shrimp2: sensitive yet practical short read mapping," *Bioinformatics*, vol. 27, pp. 1011–1012, 2011.

[28] F. Hach, F. Hormozdiari, C. Alkan, F. Hormozdiari, I. Birol, E. Eichler, and S. Sahinalp, "mrsFAST: a cache-oblivious algorithm for short-read mapping," *Nat. Methods*, vol. 7, pp. 576–577, 2010.

[29] C. Alkan, J. Kidd, T. Marques-Bonet, G. Aksay, F. Antonacci, F. Hormozdiari, J. Kitzman, C. Baker, M. Malig, O. Mutlu, and et al, "Personalized copy-number and segmental duplication maps using next-generation sequencing," *Nat. Genet.*, vol. 41, pp. 1061–1067, 2009.

[30] M. Holtgrewe, "Mason - a read simulator for second generation sequencing data," Freie Universität Berlin, Tech. Rep., 2010.

[31] 1000 genomes: a deep catalog of human genetic variation. [Online]. Available: http://www.1000genomes.org/data

[32] H. Li, B. Handsaker, A. Wysoker, T. Fennell, J. Ruan, N. Homer, G. Marth, G. Abecasis, R. Durbin, and . G. P. D. P. Subgroup, "The sequence alignment/map format and samtools," *Bioinformatics*, vol. 25, pp. 2078–2079, 2009.

[33] M. Holtgrewe, A.-K. Emde, D. Weese, and K. Reinert, "A novel and well-defined benchmarking method for second generation read mapping," *BMC Bioinformatics*, vol. 12, p. 210, 2011.