BatMis: A fast algorithm for \(k\)-mismatch mapping

Chandana Tennakoon\(^1,2\)*, Rikky W. Purbojati\(^2\)*, and Wing-Kin Sung\(^1,2\)†

\(^1\)NUS Graduate School for Integrative Sciences and Engineering, (CeLS), #05-01, 28 Medical Drive, Singapore 117456.
\(^2\)Computational Biology Lab, School of Computing, National University of Singapore. 21 Lower Kent Ridge Road, Singapore 119077.

Associate Editor: Dr. Alex Bateman

ABSTRACT
Motivation: Second Generation Sequencing (SGS) generates millions of reads that need to be aligned to a reference genome allowing errors. Although current aligners can efficiently map reads allowing a small number of mismatches, they are not well suited for handling a large number of mismatches. The efficiency of aligners can be improved using various heuristics, but the sensitivity and accuracy of the alignment is sacrificed. In this paper we introduce BatMis; an efficient method to align short reads to a reference allowing \(k\) mismatches. BatMis is a BWT based aligner that uses a seed and extend approach, and it is an exact method.

Results: Benchmark tests show that BatMis performs better than competing aligners in solving the \(k\)-mismatch problem. Furthermore, it can compete favorably even when compared with the heuristic modes of the other aligners. BatMis is a useful alternative for applications where fast \(k\)-mismatch mappings, unique mappings or multiple mappings of SGS data are required.

Availability and Implementation: BatMis is written in C/C++ and is freely available from http://code.google.com/p/batmis/

Contact: ksung@comp.nus.edu.sg

 Supplementary Information: Supplementary information is available from Bioinformatics online.

1 INTRODUCTION

Second Generation Sequencing (SGS) technologies generate a high volume of sequencing data economically and this abundance of data has introduced new possibilities to genomic studies. Applications such as whole-genome sequencing (Hillier et al., 2008), gene expression profiling (Mortazavi et al., 2008) and ChIP-seq (Mikkelsen et al., 2007) have benefited from it. All these applications need to map the SGS reads to a reference genome. Due to the differences between the sampled genome and the reference genomes and the errors introduced during the sequencing process, the mapping needs to be done allowing a reasonable number of errors. Mapping SGS reads in general require the ability to map indels. However, for platforms like Illumina and SOLiD, most of the reads can be aligned allowing mismatches only. In fact, some popular aligners like Bowtie (Langmead et al., 2009) only consider mismatches in alignment, while many others consider only mismatches by default (Lin et al., 2008; Weese et al., 2009). There are many experiments where a large number of mismatches are allowed, sometimes along with indels (Eckerle et al., 2010; Markljungh et al., 2009). Therefore, the \(k\)-mismatch problem, i.e. mapping a short read allowing \(k\)-mismatches to a reference genome is an interesting problem in bioinformatics.

Although the general \(k\)-mismatch problem can be solved heuristically with generic aligners like BLAST(Altschul et al., 1990), or exactly with aligners like BWT-SW(Lam et al., 2008), they are not practical solutions to handle tens of millions of reads produced by SGS. Therefore, specialized aligners for short read mapping are needed and the existing aligners can be broadly categorized into two classes. The first class uses a variety of hashing methods, or the indexing data structure BWT to index the reference genome (Langmead et al., 2009; Li and Durbin, 2010) . Others use hashing methods to index the reads (Li et al., 2008; Lin et al., 2008). Then, by enumerating possible mismatch patterns, the reads are aligned onto the genome. When the number of mismatches is not high, these aligners are very efficient. However, the running time will increase rapidly when the number of mismatches increases. The hashing based methods become slow since they need to look up many hash table entries as the number of allowed mismatches increases. BWT based aligners, since they simulate suffix/prefix tree traversal, become slow due to the rapid increase of branches that needs to be traversed as the number of mismatches increases. As shown in Section 3, the current aligners are slow or inadequate to handle even moderate numbers of mismatches.

To overcome the slowdown with large mismatches, aligners use various heuristic methods. A common solution is to use seeding methods (e.g. BWA, Bowtie, ELAND (Cox, 2006)). In these methods, selected seed regions of a read are aligned to the reference allowing a small number of mismatches and these alignments are extended allowing \(k\) mismatches. Some specialized methods like RazerS (Weese et al., 2009) can guarantee to find a given percentage of correct alignments. These methods cut down the alignment time dramatically. However, applying these heuristics to solve the \(k\)-mismatch problem will result in a loss of sensitivity and accuracy.

Different types of experiments require different types of mappings. The most basic type of alignment reports the first hit of a
read satisfying a given mismatch threshold. However, in some experiments, hits are required to satisfy some form of a uniqueness criterion. For example, in ChIP-seq experiments, scientists might prefer to map reads uniquely for better accuracy. Other situations require multiple hits for each read. For example, RNA-seq pipelines require multiple hits for each read. For example, RNA-seq pipelines like Tophat (Trapnell et al., 2009) need an external aligner to produce mappings of a given read, which are then post-processed for splice junctions. These pipelines usually require multiple mappings of a given read, since the first or the unique hits may map the read to pseudo-genomes or map a read covering a splice junctions to a contiguous region in the genome. Although an aligner can be designed to perform extremely well for a first hit search, it might perform relatively slow for multiple and unique mappings. Therefore, it is preferable to have a mapping algorithm that can efficiently handle these common requirements.

This paper introduces a new exact method BatMis (Basic Alignment Tool for Mismatches) that solves the $k$-mismatch problem much faster than existing methods. BatMis does not use heuristics. It is an exact method which aligns a read to the reference genome with the minimum number of mismatches. BatMis can align a read allowing up to 10 mismatches in the whole read. Our benchmarks show that BatMis is at least as fast as the current aligners. It also shows that in many cases, BatMis can compete favorably even when compared with the heuristic modes of other aligners.

2 METHODS

2.1 The $k$-mismatch problem

Let $X$ and $Y$ be two strings of equal length. The Hamming distance between $X$ and $Y$ measures the number of mismatches between $X$ and $Y$, and is denoted by $d(X, Y)$. Consider a genome $T$ and a string $P$. The $k$-mismatch problem is to find all positions $i$ such that $d(T[i..|T|], P) \leq k$. This paper is also interested in reporting the occurrences in the order of increasing mismatches, i.e., we report $i$ before $j$ if $d(T[i..|T|], P) \leq d(T[j..|T|], P)$. We have two cases.

2.2 Suffix Array and SA ranges

Suffix array is an index for exact string matching which was first introduced by (Manber and Myers, 1990). Let $T[1..n]$ be a genome of length $n$ where the nucleotides are represented by characters taken from the alphabet $\Sigma = \{a, c, g, t\}$. We assume a special character $\$" appears at the end of $T$, and it is assumed to be lexicographically smaller than all characters in $\Sigma$. We use the notation $T^\$ to denote the string constructed by reversing $T[1..n-1]$ and appending $\$" to its end. The empty string is denoted by $\epsilon$. The suffix array $SA_T[1..n]$ of $T$ is a permutation of $\{1, \ldots, n\}$ such that, for any $i < j$, the suffix starting at position $SA_T[i]$ is lexicographically smaller than the suffix starting at position $SA_T[j]$.

Let $P$ be a string. Suppose $SA_T[i]$ and $SA_T[j]$ are the lexicographically smallest and largest suffixes, respectively, having $P$ as a prefix. We define the interval $[i, j]$ as the $SA_T$ range of $P$. The length of the $SA_T$ range of $P$ is $j - i + 1$. In general, we will call $SA_T$ ranges and $SA_T$-ranges as SA-ranges.

2.3 Exact matches with BWT

The BWT, or the Burrows-Wheeler transformation (Burrows and Wheeler, 1994) $B_T$ of a string $T$ is an easily invertible permutation of $T$. $B_T$ and $SA_T$ are related by the formula $B_T[i] = T[SA_T[i]-1]$ for $SA_T[i] > 1$; and $B_T[0] = \$. Let $P$ be a substring of $T$ whose $SA_T$ range is known.

Lemma 2.1 (Backward Search). Consider a string $T$. Given $B_T$ and the $SA_T$ range $[i, j]$ of $P$, we can compute the $SA_T$ range of $P'$ in $O(1)$ time. (Ferragina and Manzini, 2000)

The result of a backward search will not be a proper interval when the $SA_T$ range does not exist. We can find the $SA_T$ range of any pattern $P$ by starting off with the empty string, whose $SA_T$ range is $[1, n]$, and compute the $SA_T$ range of $P[i..n]$ using Lemma 2.1 for $i = n$ down to 1. We call this the backward search for $P$ in $T$. We further use the backward search for $P'$ in $T^\$ to simulate the forward search for $P$ in $T$. Both types of searches will find all occurrences of the pattern $P$ in $T$, but the forward search is more natural when the pattern is searched from left to right, and the backward search is the natural choice when the pattern is searched from right to left. If an $SA_T$ range $[i, j]$ is returned after a backward search for $P$ in $T$, $P$ occurs at locations $SA_T[p]$, $p = i, \ldots, j$, in $T$. If an $SA_T$ range $[i, j]$ is returned after a forward search for $P'$ in $T$, $P$ can be found at locations $n - SA_T[p]$, $p = i, \ldots, j$, in $T$.

As an illustration, for the string $T = accatatta$, we have $B_T = atcaaatcca$, $B_{T'} = accaatc$ and

$SA_T = (9, 8, 3, 1, 4, 2, 5, 7, 6), SA_T' = (9, 8, 5, 6, 1, 7, 4, 3, 2)$.

When performing a backward search for $ac$ in $T$ by Lemma 2.1, we iteratively obtain the $SA_T$ ranges of $c$ and $ac$, which are $[6, 7]$ and $[4, 5]$, respectively. When performing a forward search for $ac$ in $T$, we will search for $ca$ in $B_{T'}$, using backward search, i.e., we iteratively obtain the $SA_T$-ranges of $a$ and $ca$, which are $[2, 5]$ and $[6, 7]$, respectively. These will translate to locations 1 and 4 in $T$.

2.4 Description of the algorithm

Consider a reference $T$. Let $R$ be a read and $K$ be a mismatch threshold. Our aim is to find the set of all strings $x$ in $T$ such that $d(x, R) \leq K$. Let $H_T^K$ be the set of all substrings $x$ in $T$ such that $d(x, R) = k$ for $k = 1, \ldots, K$. Our aim is equivalent to computing $\bigcup_{k=1}^{K} H_T^K$.

We define $R_1 = R[1..|R|/2]$ and $R_2 = R[|R|/2 + 1..|R|]$ to be the left and right halves of $R$ respectively. We propose the algorithm BatMis, which is a seed-and-extend method. It has two phases. Phase 1 finds all substrings in $T$ which look similar to $R_1$ and $R_2$ by recursion. Precisely, it computes $H_T^{R_1}$ and $H_T^{R_2}$ for a set of values of $k$ not exceeding $|K|/2$. In Phase 2, the patterns found in Phase 1 are extended to get all $k$-mismatch patterns of $R$.

The pigeon hole principle stated in Lemma 2.2 provides us with a minimal set of $H_T^{R_1}$ and $H_T^{R_2}$ guaranteed to find all $k$-mismatch patterns of $R$ with this algorithm.

Lemma 2.2. Consider a read $R$ and a string $R'$ of equal length such that $d(R, R') \leq k$, where $k \geq 1$. We have two cases.

- **Case 1:** $k$ is even. We have either $d(R'_1, R_1) \leq k/2$ or $d(R'_2, R_2) \leq k/2 - 1$.

- **Case 2:** $k$ is odd. We have either $d(R'_1, R_1) \leq (k - 1)/2$ or $d(R'_2, R_2) \leq (k - 1)/2$.

**Proof.** When $k$ is even, from the pigeon hole principle, we have the cases $d(R'_1, R_1) \leq k/2$ or $d(R'_2, R_2) > k/2$. In the first case, the proof is obvious. In the second case we have

$$d(R'_1, R_1) \leq k - d(R'_2, R_2) \leq k - k/2 - 1 = k/2 - 1.$$

Hence case 1 follows. Similarly, we can show that case 2 is true.
We can now re-state the algorithm as follows. When \( k \) is even, we extend patterns in \( \bigcup_{i=0}^{k/2} H_{R_i} \cup \bigcup_{i=0}^{k/2-1} H_{R_i} \) to obtain all \( k \)-mismatch patterns of \( R \). When \( k \) is odd, we extend patterns in \( \bigcup_{i=0}^{(k-1)/2} H_{R_i} \cup \bigcup_{i=0}^{(k-1)/2} H_{R_i} \) to obtain all \( k \)-mismatch patterns of \( R \).

The phase 2 of the algorithm where the extension of patterns are performed, is done using the procedures \( PExt \) and \( SExt \). Given a set \( X \) of substrings of \( T \), \( PExt(X, R, k) \) performs prefix extension of the strings \( x \) in \( X \) to form another set \( Y \) of strings \( y = x \cdot \alpha \) of \( T \) until every string \( y \in Y \) satisfies either (1) \( |y| = |R| \) and \( d(y, R) \leq k \) or (2) \( d(y, R[1..|y|]) = k + 1 \).

Similarly, \( SExt(X, R, k) \) performs suffix extension of the strings \( x \) in \( X \) to form another set \( Y \) of strings \( y = x \cdot \alpha \) of \( T \) until every \( y \in Y \) satisfies either (1) \( |y| = |R| \) and \( d(y, R) \leq k \) or (2) \( d(y, R[|R|-|y|+1..|R|]) = k + 1 \).

The procedures use the following recurrences and their pseudocode is shown in Figure 1.

- If \( |X| > 1 \), \( PExt(X, R, k) = \bigcup_{x \in X} PExt\{x\}, R, k \) and \( SExt\{x\}, R, k \) otherwise.

Given these two procedures, the \( k \)-mismatch patterns of \( R \) can be computed as follows. When \( k \) is even, we report \( \{x \in PExt(\bigcup_{i=0}^{k/2} H_{R_i}, R, k) \cup SExt(\bigcup_{i=0}^{k/2-1} H_{R_i}, R, k) | |x| = |R| \} = \{d(x, R) = k\} \). When \( k \) is odd, we report \( \{x \in PExt(\bigcup_{i=0}^{(k-1)/2} H_{R_i}, R, k) \cup SExt(\bigcup_{i=0}^{(k-1)/2} H_{R_i}, R, k) | |x| = |R| \} = \{d(x, R) = k\} \).

The above procedure not only computes all \( k \)-mismatch patterns of \( R \), but also reports them in the increasing order of the number of mismatches. However, it is slow since it performs a lot of redundant computations. We can modify the seed extension routine to avoid redundant computations. We divide our seed extension procedure into \( K \) iteration steps. For the \( k \)-th iteration where \( k = 0, 1, \ldots, K \), our procedure tries to obtain \( H_{R_k}^k \) i.e., all \( k \)-mismatch patterns of \( R \). In the \( 0 \)-th iteration, we start \( H_{R_0}^0 = \{R\} \) if \( R \) exists in \( T \); and \( H_{R_0}^0 = \emptyset \) otherwise. For the remaining iterations, we will not generate \( H_{R_k}^k \) starting from scratch. Instead, our routine will check all the unsuccessfully extended patterns from the \((k-1)^{\text{st}}\) iteration and see if they can be extended and become a \( k \)-mismatch pattern of \( R \). Precisely, the \( k^{\text{th}} \) iteration is divided into two stages. The first stage tries to extend those unsuccessfully extended patterns from the \((k-1)^{\text{st}}\) iteration. The second stage tries to recover the remaining \( k \)-mismatch patterns by extending a special set of seeds that guarantees to generate all the remaining \( k \)-mismatch patterns with no redundancy.

Before we give the details of Phase 2, we need some definitions to describe the set of unsuccessfully extended patterns from the \((k-1)^{\text{st}}\) iteration. By Lemma 2.2, if \( k-1 \) is odd, we need to extend the patterns in \( \bigcup_{i=0}^{(k-1)/2} H_{R_i} \cup \bigcup_{i=0}^{(k-1)/2} H_{R_i} \) to obtain all the \((k-1)^{\text{st}}\)-mismatch patterns of \( R \). If \( k-1 \) is even, we need to extend the patterns in \( \bigcup_{i=0}^{(k-1)/2} H_{R_i} \cup \bigcup_{i=0}^{(k-3)/2} H_{R_i} \) to obtain all the \((k-1)^{\text{st}}\)-mismatch patterns of \( R \). When the extended patterns accumulate \( k \) mismatches, their extensions are stopped and are marked as unsuccessfully extended patterns. These unsuccessfully extended patterns are included in \( PREC_{R_k} \) and \( SUF_{R_k} \) depending on whether they have \( k \) mismatches with a prefix or a suffix of \( R \) respectively. Formally, they are defined as follows. (Note that \( [k/2] - 1 = (k - 2)/2 \) if \( k - 1 \) is odd and \((k - 1)/2\) if \( k - 1 \) is even. Also, \( [k/2] - 1 = (k - 2)/2 \) if \( k - 1 \) is odd and \((k - 3)/2\) if \( k - 1 \) is even.)

- Let \( PRE_{R_k} \) be the set of substrings \( x \) in \( T \) such that \( d(x, R[1..|x|]) = k, x[|x|] \neq R[|R|] \) and \( d(x[1..|R|], R) \leq [k/2] - 1 \).
- Let \( SUF_{R_k} \) be the set of substrings \( x \) in \( T \) such that \( d(x, R[|R| - |x| + 1..|R|]) = k, x[|x|] \neq R[|R| - |x| + 1] \) and \( d(x[|x| - |R| + 1..|x|], R) \leq [k/2] - 1 \).

**Fig. 1.** The procedures \( PExt \) and \( SExt \) perform prefix and suffix extensions, respective, of a set of strings \( X \).
Intuitively, $PRED_k^H$ contains a subset of the shortest substrings of $T$ having exactly $k$ mismatches with a prefix of $R$, and $SUFR_k^H$ contains a subset of shortest substrings of $T$ having exactly $k$ mismatches with a suffix of $R$.

The following lemma states how to compute the sets $H_k^R$, $PRED_k^R$, and $SUFR_k^R$.

**Lemma 2.3.** Consider a read $R$. Let $P = PExt(H_0^R, R, 1)$ and $S = SExt(H_0^R \cup SUFR_1^R, R, 1)$.

a) $H_1^R = \{x \in P \cup S \mid |x, R\} = 1\}.$

b) $PRED_2^R = \{x \in P \mid d(x, R[1..|x|]) = 2\}.$

c) $SUFR_2^R = \{x \in S \mid d(x, R[R] − |x| + 1..R[|R|]) = 2\}.$

**Proof.** By Lemma 2.2, for any string $R'$ which has 1 mismatch with $R$, we have either $R'_i = R_i$ or $R'_i = R_i$. Hence, $H^R_k$ contains all strings in $P \cup S$ whose patterns have exactly 1 mismatch with $R$. The equations for $PRED_2^R$ and $SUFR_2^R$ follow by definition. □

The following two lemmas state the recursive formulas to compute $H^R_k$, $PRED_k^R$, and $SUFR_k^R$ for $k \geq 2$.

**Lemma 2.4.** Consider a read $R$ and suppose $k$ is odd. Let $P = PExt(PRED_k^R, R, k)$ and $S = SExt(H_{k/2}^R \cup SUFR_k^R, R, k)$.

a) $H_k^R = \{x \in P \cup S \mid d(x, R) = k\}.$

b) $PRED_{k+1}^R = \{x \in P \mid d(x, R[1..|x|]) = k + 1\}.$

c) $SUFR_{k+1}^R = \{x \in S \mid d(x, R[R] − |x| + 1..R[|R|]) = k + 1\}.$

**Proof.** Let $R'$ be any string in $T$ such that $d(R', R) = k$, where $k$ is odd. By Lemma 2.2, we have either (1) $d(R'_i, R'_i) \leq (k−1)/2 = (k−1)/2 \leq (k−1)/2$ or (2) $d(R'_i, R'_i) \leq (k−1)/2 = (k−1)/2 \leq (k−1)/2$ (i.e. $R'_i \subseteq H_{k/2}^R \cup SUFR_k^R, R, k$). If $R'$ satisfies (1), there should be some $\lambda$, where $|R'_\lambda| \leq \lambda \leq |R'|$, such that $d(R'[1..\lambda], R[1..\lambda]) = k$, with $R'[\lambda] \neq R[\lambda]$. By definition, $R' \in \{x \in P \mid d(x, R) = k\}$; if $R'$ satisfies (2), $R' \in \{x \in S \mid d(x, R) = k\}$. Hence, $H_k^R \subseteq \{x \in P \cup S \mid d(x, R) = k\}$. Since $H_k^R$ contains all $k$-mismatch stings of $R$ in $T$, we have $H_k^R \supseteq \{x \in P \cup S \mid d(x, R) = k\}$. From the last two relations the first identity can be obtained. By definition, $PRED_{k+1}^R$ equals

\[
\{x \in P | x \in \bigcup_{i=0}^{(k-1)/2} H_{i}^R, R, k \} | \{x \in S | d(x, R[1..|x|]) = k + 1\} = \{x \in P | d(x, R[1..|x|]) = k + 1\}.
\]

Using similar arguments, we can prove the third statement. □

**Lemma 2.5.** Consider a read $R$ and suppose $k$ is even. Let $P = PExt(H_{k/2}^R \cup PRED_k^R, R, k)$ and $S = SExt(SUFR_{k/2}^R, R, k)$.

a) $H_k^R = \{x \in P \cup S \mid d(x, R) = k\}.$

b) $PRED_{k+1}^R = \{x \in P \mid d(x, R[1..|x|]) = k + 1\}.$

c) $SUFR_{k+1}^R = \{x \in S \mid d(x, R[R] − |x| + 1..R[|R|]) = k + 1\}.$

**Proof.** The proof is similar to that of Lemma 2.4. □

Using Lemmas 2.3–2.5, Figure 2 gives the final BatMis algorithm BatMis(R, K), which computes all $k$-mismatch patterns of a read $R$ for $0 \leq k \leq K$. The first phase (lines 1 – 2) divides $R$ into two equal halves $R_l$ and $R_r$ and it recursively calls BatMis(R_l, [K/2]) and BatMis(R_r, [(K − 1)/2]) to compute $H^R_0, H^R_0$ for $0 \leq k \leq [K/2]$ and $0 \leq k \leq [(K − 1)/2]$ respectively. The second phase iteratively computes $H^R_k$ for $k = 0, \ldots, K$. It first computes $H^R_0$ (line 3), $H^R_2, PRED_2, SUFR_2$ by Lemma 2.3 (lines 5 – 7). Then, the program applies Lemmas 2.4 and 2.5 to compute ($H^R_{k+1}$, $PRED_{k+1}, SUFR_{k+1}$) iteratively for $k = 2, \ldots, K$ (see lines 8 – 17). Finally, the program reports $H^R_k$ for $k = 0, \ldots, K$.

![Fig. 2. The BatMis Algorithm.](http://bioinformatics.oxfordjournals.org/)

### 2.5 Details of implementation

We need to find all occurrences of strings corresponding to $H^R_k, H^R_{k, R}, PRED^R_k$, and $SUFR^R_k$ for $0 \leq i \leq [K/2], 0 \leq j \leq K$. Furthermore, we need an efficient way to extend them. To solve this problem, we build two BWT indexes $B^R_2$ and $B^R_T$. All strings in $H^R_k$ and $PRED^R_k$ are represented as $SA_{R^+}$-ranges, while all strings in $H^R_{k, R}$ and $SUFR^R_{k, R}$ are represented by their $SA_{R^+}$-ranges, where $0 \leq i \leq [K/2], 0 \leq j \leq K$. The extension of $SA_{R^+}$-ranges are done using forward search. Similarly, the extension of $SA_{T^-}$-ranges are done using backward search. The strings in $H^R_0$ will usually contain a mixture of $SA_{R^+}$ and $SA_{T^-}$-ranges, and it might be necessary to convert $SA_{R^+}$ ranges to $SA_{T^-}$-ranges or vice versa. This conversion is done by taking the string corresponding to $SA_{T^-}$ (or $SA_{R^+}$-) range and performing a forward (or backward) search on it. To speedup pattern searching, we also build a table containing the $SA$-ranges of all substrings of length at most 6. This table can be used to calculate the $SA$-ranges corresponding to the first few bases of a string quickly.

BatMis concatenates individual chromosomes of a genome into a single genome. The exact algorithm is run on this single genome. It might happen that a read will align to a chromosome boundary. These boundary errors occur very rarely, for example in the datasets we tested in Section 3 at most two such errors occurred per dataset. This accuracy is quite sufficient for practical purposes, but we have included a post processing script to thoroughly resolve reads with boundary errors.

The algorithm is implemented in C++. We use the BWT routines from BWT-SW (Lam et al., 2008) program. After each SA range is obtained, the corresponding locations in the genome have to be decoded. The decoding algorithm is based on (Hon et al., 2004). To facilitate the decoding, we store...
$SA_i^T$ for every $i$ which is a multiple of some fixed length $\kappa$. To compute $SA_i^T$, the algorithm counts the number of steps $s'$ needed to arrive at a sampled point $B_T[s]$ by inverting the Burrows-Wheeler transform staring at $B_T[i]$. Then, $SA_i^T = SA_i^T[s] - s'$ is computed. $SA_i^T$ needs to be sampled as $SA_i^T$-ranges need to be decoded as well. We improve this decoding step further in the following way. During the extension step, if the string being extended occurs uniquely in T and the corresponding $SA_i$-range was sampled, we save this $SA_i$-range. We next count the number of steps $s'$ needed to complete the extension. With this information, the location in the genome can be calculated using the formula above. In the actual implementation, since storing the sampled $SA_i$-ranges take a lot of memory, the implementation can optionally convert all $SA_i$ ranges to $SA_i$-ranges (or vice versa) and use only one sampling. For the human genome, if only one sampling is used with sampling length $\kappa = 8$, the decoding algorithm can be run under 4GB of RAM.

Furthermore, the recursions are unrolled for efficiency. For mismatch thresholds less than 5, the algorithm is implemented as stated. When scans are performed allowing a large number of mismatches, storing $SU_{R,\kappa}$ and $PRE_{R,\kappa}$ requires a lot of memory. To reduce memory usage, for $k > 5$, the set of $k$-mismatch hits, $H^k_{R,\kappa}$, is computed directly based on Lemma 2.2. Although this approach reduces the memory required to store $SU_{R,\kappa}$ and $PRE_{R,\kappa}$ for $i > 5$, it will also generate duplicate hits. Post-processing steps are performed to remove the duplicate hits.

When mapping SOLiD reads, the reference genome is converted to color space. To convert color space reads to nucleotide space, the algorithm given in BWA is used.

3 RESULTS

There are a vast number of sequence aligners that can perform exact $k$-mismatch alignment. Different aligners have different policies regarding how to rank the hits and how to handle uncalled bases (e.g. N), which makes straightforward comparison difficult. To allow us to compare different aligners fairly, we use data sets without uncalled bases. We test each program’s ability to report the least mismatch hits, the unique hits and multiple hits. By the least mismatch hit of a read $R$ in the reference $T$, we mean any position $i$ such that $d(R, T[i..i + |R| - 1]) \leq k$, and for any other position $j$ we have $d(R, T[j..j + |R| - 1]) \geq d(R, T[i..i + |R| - 1])$. By a unique hit of a read $R$ in the reference $T$, we mean a unique lowest mismatch hit of $R$ in $T$.

For comparison with our algorithm, we chose BWA (version 0.5.9-r16) to represent aligners of the BWT family, RazerS2 to represent $q$-gram methods and ZOOM (Linux64 demo version 1.5.5.20120225072719) to represent methods that use gapped seeds. We believe that these programs are among the best in the literature.

3.1 Ability to detect mismatches.

In this section we examine the robustness of mismatch mappings of the selected aligners. We randomly extracted two sets of 51bp and 100bp reads from regions of hg18. Each dataset contained 100,000 reads. New $k$-mismatch datasets were created by introducing exactly $k$ mismatches uniformly at random to all reads in the original dataset for $k = 0, 1, 2, 3, 4, 5, 8, 10$. If an aligner performs $k$-mismatch mapping correctly, it must be able to map all $k$-mismatch reads to the reference genome allowing $k$-mismatches. Supplementary Table 1 summarizes these results. BWA missed some hits with a large number of mismatches for 100bp reads. It was able to map all the reads with up to five mismatches, but was only able to map 97291 and 15124 of reads having 8 and 10 mismatches respectively. Other aligners were able to map back all the reads. This result suggests that, BatMis, ZOOM and Razers2 can detect $k$-mismatches effectively but BWA might miss some hits when $k$ is large.

3.2 Mapping real data

This section studies the performance of different algorithms using real data. We first check the performance of each aligner when reporting the least mismatch hits. Many biologists prefer to have unique hits as a criteria to filter out noise. Therefore, we also measure the performance of different aligners on finding unique hits in real data.

The evaluation used the Illumina sequencing datasets ERR000577 and ERR024201 taken from the European Nucleotide Archive. The datasets contained reads of lengths 51bp and 100bp respectively. These datasets are paired end reads, and we chose the datasets containing the forward reads. Reads containing uncalled bases were filtered out, and the first 1,000,000 reads were selected for benchmarking. These sets of reads were mapped to the reference genome hg18 allowing different numbers of mismatches. ZOOM and Razers2 produce a small number of false mappings. The results of these mappings, are given in Tables 1 and 2.

In general, all aligners report a similar number of hits for both 51bp and 100bp reads. However, BWA will report significantly less number of hits compared to the other aligners when the number of mismatches is large. For all the reads where another aligner can find a least mismatch hit, BatMis will also report a least mismatch hit. Also, BatMis will report all the correct unique hits found by other aligners.
different numbers of mismatches. Entries in bold produce false hits, and the number of false hits is shown inside the brackets. ZOOM, RazerS2 and BW A were run in their exact modes.

<table>
<thead>
<tr>
<th>51bp</th>
<th>0-mis</th>
<th>1-mis</th>
<th>2-mis</th>
<th>3-mis</th>
<th>4-mis</th>
<th>5-mis</th>
<th>Time (s)</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td># of Hits</td>
<td>Time (s)</td>
<td># of Hits</td>
<td>Time (s)</td>
<td># of Hits</td>
<td>Time (s)</td>
<td># of Hits</td>
</tr>
<tr>
<td>BatMis</td>
<td>639252</td>
<td>11</td>
<td>833631</td>
<td>19</td>
<td>905121</td>
<td>31</td>
<td>940232</td>
</tr>
<tr>
<td>BWA</td>
<td>639252</td>
<td>43</td>
<td>833631</td>
<td>64</td>
<td>905121</td>
<td>179</td>
<td>940232</td>
</tr>
<tr>
<td>ZOOM</td>
<td>639252</td>
<td>1007</td>
<td>833631</td>
<td>1152</td>
<td>905121</td>
<td>1731</td>
<td>9402323</td>
</tr>
<tr>
<td>RazerS2</td>
<td>639252</td>
<td>16240</td>
<td>833631</td>
<td>17083</td>
<td>905121</td>
<td>16845</td>
<td>940232</td>
</tr>
</tbody>
</table>

Table 1. Statistics for finding least mismatch hits of 1,000,000 reads taken from the 51bp dataset ERR000577 and the 100bp dataset ERR0252401 allowing different numbers of mismatches. Entries in bold produce false hits, and the number of false hits is shown inside the brackets. ZOOM, RazerS2 and BW A were run in their exact modes.

<table>
<thead>
<tr>
<th>51bp</th>
<th>0-mis</th>
<th>1-mis</th>
<th>2-mis</th>
<th>3-mis</th>
<th>4-mis</th>
<th>5-mis</th>
<th>Time (s)</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td># of Hits</td>
<td>Time (s)</td>
<td># of Hits</td>
<td>Time (s)</td>
<td># of Hits</td>
<td>Time (s)</td>
<td># of Hits</td>
</tr>
<tr>
<td>BatMis</td>
<td>586314</td>
<td>13</td>
<td>759079</td>
<td>21</td>
<td>817844</td>
<td>38</td>
<td>843466</td>
</tr>
<tr>
<td>BWA</td>
<td>586314</td>
<td>43</td>
<td>759079</td>
<td>64</td>
<td>817844</td>
<td>179</td>
<td>843466</td>
</tr>
<tr>
<td>ZOOM</td>
<td>586314</td>
<td>1069</td>
<td>759079</td>
<td>1190</td>
<td>817844</td>
<td>1764</td>
<td>8434671</td>
</tr>
<tr>
<td>RazerS2</td>
<td>586314</td>
<td>15689</td>
<td>759079</td>
<td>16308</td>
<td>817844</td>
<td>16562</td>
<td>8434682</td>
</tr>
</tbody>
</table>

Table 2. Statistics for finding multiple hits of 1,000,000 reads taken from the 51bp dataset ERR000577 and 100bp dataset ERR0252401 allowing different numbers of mismatches. Entries in bold produce false hits, and the number of false hits is shown inside the brackets. ZOOM, RazerS2 and BW A were run in their exact modes.

<table>
<thead>
<tr>
<th>100bp</th>
<th>2-mis</th>
<th>3-mis</th>
<th>4-mis</th>
<th>5-mis</th>
<th>Time (s)</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td># of Hits</td>
<td>Time (s)</td>
<td># of Hits</td>
<td>Time (s)</td>
<td># of Hits</td>
</tr>
<tr>
<td>BatMis</td>
<td>871712</td>
<td>39</td>
<td>881586</td>
<td>44</td>
<td>887947</td>
</tr>
<tr>
<td>BWA</td>
<td>871712</td>
<td>193</td>
<td>881586</td>
<td>257</td>
<td>887947</td>
</tr>
<tr>
<td>ZOOM</td>
<td>871712</td>
<td>1597</td>
<td>881586</td>
<td>2065</td>
<td>887947</td>
</tr>
<tr>
<td>RazerS2</td>
<td>871712</td>
<td>31580</td>
<td>881586</td>
<td>30195</td>
<td>8879481</td>
</tr>
</tbody>
</table>

Table 3. Statistics for finding multiple hits of 1,000,000 reads taken from the 100bp dataset ERR0252401 allowing different numbers of mismatches. Entries in bold produce false hits, and the number of false hits is shown inside the brackets. ZOOM, RazerS2 and BW A were run in their exact modes.

<table>
<thead>
<tr>
<th>51bp</th>
<th>2-mis</th>
<th>3-mis</th>
<th>4-mis</th>
<th>5-mis</th>
<th>Time (s)</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td># of Hits</td>
<td>Time (s)</td>
<td># of Hits</td>
<td>Time (s)</td>
<td># of Hits</td>
</tr>
<tr>
<td>BatMis</td>
<td>817844</td>
<td>38</td>
<td>843466</td>
<td>55</td>
<td>856383</td>
</tr>
<tr>
<td>BWA</td>
<td>817844</td>
<td>191</td>
<td>814850467</td>
<td>397</td>
<td>8528751400</td>
</tr>
<tr>
<td>RazerS2</td>
<td>816536119</td>
<td>10567</td>
<td>842838107</td>
<td>10669</td>
<td>855989185</td>
</tr>
</tbody>
</table>

Table 4. Statistics for finding multiple hits of 1,000,000 reads taken from the 51bp dataset ERR000577 and 100bp dataset ERR0252401 allowing different numbers of mismatches. BWA and RazerS2 were run in their heuristic modes. Entries in bold produce false hits, and the number of false hits is shown inside the brackets.
3.3 Multiple mappings

We benchmarked the time taken by each aligner to produce multiple mappings of a given read. We used the 100bp real life data set that was used in the previous section. BatMis and BWA have options to scan all possible hits with less than \( k \) mismatches. For ZOOM and RazerS2, since such a mode is not present, the mapping was done to produce the first 100,000 hits for a given read. ZOOM and BWA gave a small number of false hits mainly due to boundary errors. The results are given in Table 3. BatMis reports all the true hits found by other aligners and is faster than them when performing multiple mappings. All aligners report about the same number of hits, although RazerS2 and BWA reports less number of hits compared with the other aligners when the number of mismatches is large.

3.4 Comparison against heuristic methods

Instead of searching for the exact solution, BWA and RazerS2 can employ heuristics to speed up mapping. BWA will first find hits in a seed region allowing at most two mismatches, and extend the rest of the read allowing a given number of mismatches in the full read. RazerS2 has a heuristic mode where the reads can be mapped with 99% accuracy. Heuristics may miss some hits. This will result in incorrectly calling uniquely mapped reads. Table 4 shows the speed and the number of true unique hits recovered using heuristics modes of BWA and RazerS2 against the exact algorithm of BatMis. The mapping procedure was similar to that in Section 3.2, except that BWA was run in its seeding mode and RazerS2 was run with its default sensitivity of 99%.

The results show that BatMis is much faster than RazerS2 in all cases. BWA performs very well in its seeded mode on real data for long reads. For 51bp reads, BatMis is faster than BWA and produces more mappings. For 100bp reads, BatMis is faster than BWA for up to 5-mismatches. At 8 and 10 mismatches the speeds are similar, with BatMis again producing more hits. The false unique hits reported by the aligners in their heuristic modes are negligible, and a reasonable number of correct unique hits were recovered. Supplementary table 2 shows the statistics for obtaining all \( k \)-mismatch hits with BWA in its heuristic mode and with BatMis. RazerS2 was not considered as it is slow even for finding unique hits. We can see that for both types of reads, BatMis is faster or has comparable speed with BWA. The heuristics of BWA can produce exact results up to two mismatches. For the 100bp reads, BWA misses between 2%-9% hits, and for 51bp reads it will miss 11%-30% hits when there are 3 to 5 mismatches.

4 DISCUSSION

The solution for the \( k \)-mismatch mapping is important to Second Generation Sequencing. We introduced a new algorithm BatMis that can solve the \( k \)-mismatch problem exactly and efficiently. We checked the ability to find least mismatch hits, unique hits and multiple hits of some of the current state of the art aligners. Our results show that some aligners cannot reliably map reads with a large number of mismatches. On the other hand, BatMis was able to recover all the hits and was faster. Finally, BatMis is faster or has a comparable performance with the heuristic methods of other aligners. These results show that BatMis is a robust aligner that performs well at all mismatch thresholds. One limitation of BatMis is that it cannot handle paired-end reads and indels. We believe that BatMis is a useful alternative for mapping SGS reads when we want to perform multiple mapping, unique mapping, or when we want to tolerate a large number of mismatches.

5 ACKNOWLEDGEMENT

This work was supported in part by the the MOEs AcRF Tier 2 funding R-252-000-444-112.

REFERENCES