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), #06-01, 28 Medical Drive, Singapore 117456 and $^2$Computational Biology Lab, School of Computing, National University of Singapore, 21 Lower Kent Ridge Road, Singapore 119077, Singapore

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 alignments are sacrificed. In this article, we introduce Basic Alignment tool for Mismatches (BatMis)—an efficient method to align short reads to a reference allowing $k$ mismatches. BatMis is a Burrows-Wheeler transformation 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 models 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: sungt@comp.nus.edu.sg

Supplementary Information: Supplementary information is available from Bioinformatics online.

Received on December 6, 2011; revised on May 27, 2012; accepted on June 6, 2012

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 [Li et al. (2008); Weese et al. (2010)].

There are many experiments where a large number of mismatches are allowed, sometimes along with indels [Eckerle et al. (2012); Marklund et al. (2012)]. 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 [Wheeler et al. (1994)] or exactly with aligners like BWT-SW [Lan 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 [Lan et al. (2008); Li and Durbin (2011)]. Others use hashing methods to index the reads [Lin et al. (2010); Li et al. (2010)].

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 2, 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 and 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 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-genes 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 article introduces a new exact method BatMis (Basic Affine Gapless) (Saulnier and Myers, 2002) that solves the k-mismatch problem much faster than existing methods. BatMis does not use heuristics. It is an exact method that 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 R. The k-mismatch problem is to find all positions i such that d(R, T[i−|R|+1:i]) ≤ k.

This article is also interested in reporting the occurrences in the order of increasing mismatches, i.e., we report i before j if d(R, T[i−|R|+1:i]) < d(R, T[j−|R|+1:j]).

2.2 Suffix array and SA ranges

Suffix array is an index for exact string matching which was first introduced by Gusfield and Myers (1992). Let T[1..n] be a genome of length n where the nucleotides are represented by characters taken from the alphabet Σ = {a, c, g, t}. We assume a special character $\epsilon$ 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 $\epsilon$ to its end. The empty string is denoted by $\epsilon$. The suffix array $SA[1..n]$, of T is a permutation of {1, n} such that, for any i < j, the suffix starting at position $SA[i]$ is lexicographically smaller than the suffix starting at position $SA[j]$.

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

2.3 Exact matches with BWT

The BWT, or the Burrows–Wheeler transformation (Burrows and Wheeler, 1994) of a string T is an easily invertible permutation of T. By and $SA[T]$ are related by the formula $By[SA[T][i]] = T[SA[T][i]]$ for $SA[T][1]..SA[T][n]$, and $By[1] = 1$. Let P be a substring of T whose SA range is known.

Lemma 2.1 (Backward Search). Given a string T. Given By and the SA range [i, j] of P, we can compute the SA range of P in O(|P|) time (Ferragina and Manzini, 2000).

The result of a backward search will not be a proper interval when the SA range does not exist. We can find the SA range of any pattern P by starting off with the empty string, whose SA range is [1, n], and compute the SA range of $P \cdot \epsilon$ using Lemma 2.1 before $\epsilon$ is 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 range $[i, j]$ is returned after a backward search for P in T, P occurs at locations $SA[p][j]$ for $p = i..j$ in T. If an $SA[p][j]$ range $[i, j]$ is returned after a forward search for P in T, P can be found at locations $n−SA[p][j]$ for $p = i..j$. In T.

As an illustration, for the string $T = acaactta$, we have

$By_1 = \text{null array}; By_2 = \text{null array}$ and

$SA_{T} = (9, 8, 3, 1, 4, 2, 5, 7, 6); S_{T} = (9, 8, 5, 6, 1, 7, 4, 3, 2)$. When performing a backward search for $ac$ in T by Lemma 2.1, we iteratively obtain the SA 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 By−1 using backward search, i.e., we iteratively obtain the SA ranges of $ca$, and $ac$, 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 $d(x, R) ≤ k$. We call $HT_0$ the set of all substrings $x$ in T such that $d(x, R) = k$ for $k = 1..K$. Our aim is equivalent to computing $\bigcup_{k=0}^{K} HT_k$.

We define $R_0 = R[1:|R|/2]$ and $R_1 = R[|R|/2+1:|R|]$ 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 $R_0$ and $R_1$ by recursion. Precisely, it computes $HT_0$ and $HT_1$, 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 $HT_0$ and $HT_1$ guaranteed to find all k-mismatch patterns of R with this algorithm.

Lemma 2.2. Consider a read R and a string K of equal length such that $d(R, K) ≤ k$, where $k ≥ 1$. Let $HT_0$ be the set of all substrings x in T such that $d(x, R) ≤ k$. We have two cases.

• Case 1: k is even. We have either $d(R_0, K) ≤ k/2$ or $d(R_1, K) ≤ k/2 − 1$.
• Case 2: k is odd. We have either $d(R_0, K) ≤ (k−1)/2$ or $d(R_1, K) ≤ (k−1)/2$.

Proof. When k is even, from the pigeon hole principle, we have the cases $d(R_0, K) ≤ k/2$ or $d(R_1, K) ≥ k/2$. In the first case, the proof is obvious. In the second case we have

$$d(R_1, K) ≤ k−d(R_0, K) ≤ k−k/2−1=k/2.$$ 

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 $\bigcup_{k=1}^{K} HT_0 \cup \bigcup_{k=1}^{K} HT_1$ to obtain all k-mismatch patterns of R. When k is odd, we extend patterns in $\bigcup_{k=1}^{K−1} HT_0 \cup \bigcup_{k=1}^{K−1} HT_1$ 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 of X of T until every string y of X satisfies either (1) $|y| \geq |R|$ and $d(y, R) \leq k$ or (2) $d(y, R[|R|−|y|+1:|R|]) = k+1$. Similarly, $SExt(x, R, k)$ performs suffix extension of the strings x in T to form another set Y of strings $y = x \circ |x|/2$ of T until every string y of X satisfies either (1) $|y| = |R|$ and $d(y, R) ≤ k$ 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 $|X| > 1$, $PExt(X, R, k) = \bigcup_{x \in X} PExt(x, R, k)$
We divide our seed extension procedure into can modify the seed extension routine to avoid redundant computations. 

\[ \text{Intuitively, } \text{PRE}_k \text{ computed as follows. When } R \text{ mismatch patterns of } k \text{ in even, we report } (x \in \text{PRE}_k)(u \subseteq H_k, R, k) \text{, otherwise. When } k \text{ is odd, we report } (x \in \text{PRE}_k)(u \subseteq H_k, R, k) \text{, otherwise.} \]

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 \( k + 1 \) iterations. For the \( kth \) iteration where \( k = 1, 2, \ldots, K \), our procedure tries to obtain \( H_k \), i.e. all \( k \)-mismatch patterns of \( R \). In the \( kth \) iteration, we set \( H_k = \emptyset \) if \( R \) exists in \( T \), and \( H_k = R \) otherwise. For the remaining iterations, we will not generate \( H_k \) starting from scratch. Instead, our routine will check all the unsuccessfully extended patterns from the \( (k-1)th \) iteration and see if they can be extended and become a \( k \)-mismatch pattern of \( R \). Precisely, the \( kth \) iteration is divided into two steps. The first stage tries to extend those unsuccessfully extended patterns from the \( (k-1)th \) 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 \( (k-1)th \) iteration. By Lemma 2.3 if \( k = 1 \) is odd, we need to extend the patterns in \( \bigcup_{i=1}^{k-2} H_{\frac{i}{2}} \cup \bigcup_{i=k+2}^{k} H_{\frac{i}{2}} \) to obtain the \( (k-1) \)-mismatch patterns of \( R \). If \( k = 1 \) is even, we need to extend the patterns in \( \bigcup_{i=1}^{k-2} H_{\frac{i}{2}} \cup \bigcup_{i=1}^{k-2} H_{\frac{i}{2}} \) to obtain the \( (k-1) \)-mismatch patterns of \( R \). When the extended patterns accumulate \( k \)-mismatch patterns, their extensions are stopped and are marked as unsuccessfully extended patterns. These unsuccessfully extended patterns are included in \( \text{PRE}_k \) and \( \text{SUP}_k \) depending on whether they have \( k \) mismatches with a prefix or a suffix of \( R \), respectively. Formally, they are, as defined below. (Note that \( (k-2i) \)-mismatch patterns if \( k = 1 \) is odd and \( (k-2i) \)-mismatch patterns if \( k = 1 \) is even.) Let \( \text{PRE}_k \) be a subset of substrings \( x \) in \( T \) such that \( d(x, [R[1:|x|] = k \text{ and } x[|x|] \neq R[|x|]) \) and \( d(x, [R[1:|x|] = 1 \text{ and } x[|x|] = R[|x|]) \) if \( k = 1 \) is odd and \( k < 3 \) if \( k = 1 \) is even.) Let \( \text{SUf}_k \) be a subset of substrings \( x \) in \( T \) such that \( d(x, [R[1:|x|] = k \text{ and } x[|x|] \neq R[|x|]) \) and \( d(x, [R[1:|x|] = 1 \text{ and } x[|x|] \neq R[|x|]) \) if \( k = 1 \) is odd and \( k < 3 \) if \( k = 1 \) is even.)

Intuitively, \( \text{PRE}_k \) contains a subset of the shortest substrings of \( T \) having exactly \( k \) mismatches with a prefix of \( R \) and \( \text{SUf}_k \) 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 \( H_k \), \( \text{PRE}_k \), and \( \text{SUf}_k \).

**Lemma 2.3.** Consider a read \( R \). Let \( P = \text{PRE}(R, k) \) and \( S = \text{SUf}(R, k) \). Then
a) \( H_k = (x \in \text{PRE}(R, k)) \).

**Proof.** By Lemma 2.2 for any string \( R \) which has 1 mismatch with \( R \), we have either \( R \) or \( R \) contains all strings in \( \text{PRE}(R, k) \) whose patterns have exactly 1 mismatch with \( R \). The equations for \( \text{PRE}_k \) and \( \text{SUf}_k \) follow by definition.

**Fig. 1.** The procedures \( \text{PRE} \) and \( \text{SUf} \) perform prefix and suffix extensions, respectively, of a set of strings \( \mathcal{X} \).

The following two lemmas state the recursive formulas to compute \( H_k \), \( \text{PRE}_k \) and \( \text{SUf}_k \) for \( k \geq 2 \).

**Lemma 2.4.** Consider a read \( R \) and suppose \( k \) is odd. Let \( P = \text{PRE}(R, k) \) and \( S = \text{SUf}(R, k) \). Then
a) \( H_k = (x \in \text{SUf}(R, k)) \).

**Proof.** By Lemma 2.2 for any string \( R \) which has 1 mismatch with \( R \), we have either \( R \) or \( R \) contains all strings in \( \text{SUf}(R, k) \) whose patterns have exactly 1 mismatch with \( R \). The equations for \( \text{PRE}_k \) and \( \text{SUf}_k \) follow by definition.
second phase iteratively computes $H_k$.

The strings in $BT$ applies Lemmas 2.4 and 2.5 to compute ($\ast$).

If $R$ satisfies (1), there should be some $\lambda$, where $|R| \leq \lambda \leq |R|$, such that $|H_{R(1)}R(1)|=\lambda$. From Lemma 2.3, $R \in \text{P}(d(x,R))$.

By definition, $R \in \text{P}(d(x,R)) \iff R \in \text{P}(d(x,R)|\lambda|)$. Hence, $H_k=\bigcup \{x \in \text{P}(d(x,R))\}$.

Since $H_k$ contains all $k$-mismatch strings of $R \in \mathcal{R}$, we have $H_k=\bigcup \{x \in \text{P}(d(x,R)|\lambda|)\}$. From the last two relations, the first identity can be obtained.

By definition, $\text{PRE}_k^{+1}$ equals:

$$\begin{align*}
(\{x \in \text{P}(d(x,R)|\lambda|) \mid |\lambda-k| \geq 1\} & \cup \{x \in \text{P}(|d(x,R)|-k\) \mid |\lambda-k| \geq 1\}
= \{x \in \text{P}(|d(x,R)|-k\) \mid |\lambda-k| \geq 1\}
= \{x \in \text{P}(|d(x,R)|-k\) \mid |\lambda-k| \leq 1\}
\end{align*}$$

Using similar proofs, we can prove the third statement.

**Lemma 2.5.** Consider a read $R$ and suppose $k$ is even. Let $P = \text{P}(\text{PRE}_k^{+1}(R)) \cap \text{PRE}_k(R)$, and $S = \text{SET}(\text{PRE}_k(R), k)$.

$a)$ $H_k=\bigcup \{x \in \text{P}(d(x,R)|\lambda|) \mid |\lambda-k| \leq 1\}$

$b)$ $\text{PRE}_k^{+1}=\{x \in \text{P}(|d(x,R)|-k) \mid |\lambda-k| \leq 1\}$

$c)$ $\text{SUF}_k^{+1}=\{x \in \text{P}(|d(x,R)|-k) \mid |\lambda-k| \leq 1\}$

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

Using Lemmas 2.4 and 2.5, Figure 2 gives the final BatMis algorithm.

**BatMis**($R$, $K$)

**Ensure:** Report $H_k$ for $k = 0, 1, \ldots, K$.

**Phase 1:** Compute $H_k$ for $0 = 0, 1, \ldots, K/2$)

1. $H_0 = H_1 = \ldots = H_0$.

2. Compute ($H_k$, for $0 = 0, 1, \ldots, (K/2)$) by calling $\text{BatMis}(R, \{K/2\})$.

3. Set $H_k = (R)$ if $R \in T$ and $i$ otherwise.

4. for $k = 1$ to $K$ do

5. if $k = 1$ then

6. $H_1 = \text{PExt}(H_k(R, \{R\})$;

7. $S = \text{SE}(H_k(R, \{R\})$;

8. else if $i$ is odd then

9. $P = \text{PExt}(H_k(R, \{R\})$;

10. $S = \text{SE}(H_k(R, \{R\})$;

11. else

12. $P = \text{PExt}(H_k(R, \{R\})$;

13. $S = \text{SE}(H_k(R, \{R\})$;

14. end if

15. $H_k = \{x \in \text{P} \mid d(x, R) = k, \lambda = |R|\}$;

16. $\text{PRE}_k^{+1} = \{x \in \text{P} \mid d(x, R) = k, \lambda = |R|\}$;

17. $\text{SUF}_k^{+1} = \{x \in \text{P} \mid d(x, R) = k+1, \lambda = |R|\}$;

18. end for

19. Return ($H_k$, for $k = 0, 1, \ldots, K$).

**Fig. 2.** The BatMis algorithm.

SA$[i]$, the algorithm counts the number of steps $s'$ needed to arrive at a sampled point $B_s[i]$ by inverting the Burrows–Wheeler transform starting at $B_s[i]$. Then, SA$[i]$ is SA$[s']-s'$ computed. SA$[i]$ needs to be sampled as SA$[s']$ 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-range was sampled, we save this SA-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-ranges takes a lot of memory, the implementation can optionally convert all SA$[s']$ ranges to SA$[s']$ ranges (or vice versa) and use only one sampling. For the human genome, if only one sampling is used with sampling length $k=8$, the decoding algorithm can be run under 4 GB 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 $\text{PRE}_k$ and $\text{SUF}_k$ requires a lot of memory. To reduce memory usage, for $k > 5$, the set of $k$-mismatch hits, $H_k$, is computed directly based on Lemma 2.4. Although this approach reduces the memory required to store $\text{PRE}_k$ and $\text{SUF}_k$ for $i > 5$, it will also generate duplicate hits. Post-processing steps are performed to remove the duplicate hits.

When mapping SOLO 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...
Table 1. Statistics for finding least mismatch hits of 1,000,000 reads taken from the 51 bp dataset ERR000577 and the 100 bp dataset ERR024201 allowing different numbers of mismatches

<table>
<thead>
<tr>
<th></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>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>No. of Hits</td>
<td>Time (s)</td>
<td>No. of Hits</td>
<td>Time (s)</td>
<td>No. of Hits</td>
<td>Time (s)</td>
</tr>
<tr>
<td>51 bp</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>BatMis</td>
<td>639 252</td>
<td>11</td>
<td>833 631</td>
<td>19</td>
<td>905 121</td>
<td>31</td>
</tr>
<tr>
<td>BWA</td>
<td>639 252</td>
<td>43</td>
<td>833 631</td>
<td>64</td>
<td>905 121</td>
<td>179</td>
</tr>
<tr>
<td>ZOOM</td>
<td>639 252</td>
<td>1007</td>
<td>833 631</td>
<td>1152</td>
<td>905 121</td>
<td>1731</td>
</tr>
<tr>
<td>RazerS2</td>
<td>639 252</td>
<td>16240</td>
<td>833 631</td>
<td>17 083</td>
<td>905 121</td>
<td>16 845</td>
</tr>
<tr>
<td>100 bp</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>BatMis</td>
<td>908 432</td>
<td>32</td>
<td>920 354</td>
<td>37</td>
<td>928 493</td>
<td>60</td>
</tr>
<tr>
<td>BWA</td>
<td>908 432</td>
<td>193</td>
<td>920 354</td>
<td>257</td>
<td>928 493</td>
<td>574</td>
</tr>
<tr>
<td>ZOOM</td>
<td>908 432</td>
<td>1612</td>
<td>920 354</td>
<td>2160</td>
<td>928 493</td>
<td>2418</td>
</tr>
<tr>
<td>RazerS2</td>
<td>908 432</td>
<td>340 04</td>
<td>920 354</td>
<td>328 92</td>
<td>928 493</td>
<td>327 87</td>
</tr>
</tbody>
</table>

Entries in bold produce false hits, and the number of false hits is shown inside the brackets. ZOOM, RazerS2 and BWA were run in their exact modes.

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 51 bp and 100 bp 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 and 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 100 bp reads. It was able to map all the reads with up to five mismatches, but was only able to map 97,291 and 15,124 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 51 bp and 100 bp, 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 H and I.
The results are given in Table 3. BatMis reports all the true hits. We benchmarked the time taken by each aligner to produce multiple mappings. All aligners report about the same number of hits found by other aligners and is faster than them when performing multiple mappings. All aligners report about the same number of hits found by other aligners and is faster than them when performing multiple mappings.

3.3 Multiple mappings

We benchmarked the time taken by each aligner to produce multiple mappings. All aligners report about the same number of hits found by other aligners and is faster than them when performing multiple mappings. All aligners report about the same number of hits found by other aligners and is faster than them when performing multiple mappings.

### Table 2. Statistics for finding the unique hits of 1,000,000 reads taken from the 51 bp dataset ERR000577 and 100 bp dataset ERR024201 allowing different numbers of mismatches

<table>
<thead>
<tr>
<th>Program</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>
</tr>
</thead>
<tbody>
<tr>
<td>BatMis</td>
<td>12 709 391</td>
<td>30 000</td>
<td>31 000</td>
<td>195 000</td>
<td>260 000</td>
<td>315 000</td>
</tr>
<tr>
<td>BWA</td>
<td>12 709 417(26)</td>
<td>172 000</td>
<td>411 000</td>
<td>59 121 789(94)</td>
<td>260 000</td>
<td>143 186 148(168)</td>
</tr>
<tr>
<td>ZOOM</td>
<td>12 709 421(30)</td>
<td>1581 000</td>
<td>381 000</td>
<td>59 121 802(187)</td>
<td>2307 000</td>
<td>163 479 167</td>
</tr>
<tr>
<td>RazerS2</td>
<td>12 709 391</td>
<td>30 778</td>
<td>31 000 921</td>
<td>31 956 591 193 338</td>
<td>32 340 96 498 204</td>
<td>32 747 143 309 572</td>
</tr>
</tbody>
</table>

Entries in bold produce false hits, and the number of false hits is shown inside the brackets. ZOOM, RazerS2 and BWA were run in their exact modes.

### Table 3. Statistics for finding multiple hits of 1,000,000 reads taken from the 100 bp dataset ERR024201 allowing different numbers of mismatches

<table>
<thead>
<tr>
<th>Program</th>
<th>1-mis</th>
<th>2-mis</th>
<th>3-mis</th>
<th>4-mis</th>
<th>5-mis</th>
</tr>
</thead>
<tbody>
<tr>
<td>BatMis</td>
<td>12 709 391</td>
<td>30 000</td>
<td>31 000</td>
<td>195 000</td>
<td>260 000</td>
</tr>
<tr>
<td>BWA</td>
<td>12 709 417(26)</td>
<td>172 000</td>
<td>411 000</td>
<td>59 121 789(94)</td>
<td>260 000</td>
</tr>
<tr>
<td>ZOOM</td>
<td>12 709 421(30)</td>
<td>1581 000</td>
<td>381 000</td>
<td>59 121 802(187)</td>
<td>2307 000</td>
</tr>
<tr>
<td>RazerS2</td>
<td>12 709 391</td>
<td>30 778</td>
<td>31 000 921</td>
<td>31 956 591 193 338</td>
<td>32 340 96 498 204</td>
</tr>
</tbody>
</table>

Entries in bold produce false hits, and the number of false hits is shown inside the brackets. ZOOM, RazerS2 and BWA were run in their exact modes.

In general, all aligners report a similar number of hits for both 51 bp and 100 bp reads. However, BWA will report significantly less number of hits compared with 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. In addition, BatMis will report all the correct unique hits found by other aligners.

### 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. 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 51 bp reads, BatMis is faster than BWA and produces more mappings. For 100 bp reads, BatMis is faster than BWA for up to five 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.
The solution for the $k$-mismatch problem exactly and efficiently. We tolerate a large number of mismatches. BatMis 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.

#### 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 comparable performance with the heuristic methods of other aligners. These results show that some aligners cannot reliably map reads with a large number of mismatches.

**Table 4. Statistics for finding the unique hits of 1 000 000 reads taken from the 51 bp dataset ERR000577 and 100 bp dataset ERR024201 allowing different numbers of mismatches**

<table>
<thead>
<tr>
<th>Program</th>
<th>2-mis</th>
<th>3-mis</th>
<th>4-mis</th>
<th>5-mis</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>No. of Hits</td>
<td>Time (s)</td>
<td>No. of Hits</td>
<td>Time (s)</td>
</tr>
<tr>
<td>BatMis</td>
<td>817 844</td>
<td>38</td>
<td>843 466</td>
<td>55</td>
</tr>
<tr>
<td>BWA</td>
<td>817 844</td>
<td>191</td>
<td>841 850 (467)</td>
<td>397</td>
</tr>
<tr>
<td>RazerS2</td>
<td>816 536 (119)</td>
<td>10 567</td>
<td>822 838 (107)</td>
<td>10 669</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Program</th>
<th>100 bp</th>
<th>2-mis</th>
<th>3-mis</th>
<th>4-mis</th>
<th>5-mis</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>No. of Hits</td>
<td>Time (s)</td>
<td>No. of Hits</td>
<td>Time (s)</td>
<td>No. of Hits</td>
</tr>
<tr>
<td>BatMis</td>
<td>881 586</td>
<td>44</td>
<td>892 698</td>
<td>96</td>
<td>903 577</td>
</tr>
<tr>
<td>BWA</td>
<td>817 844</td>
<td>191</td>
<td>887 122 (167)</td>
<td>484</td>
<td>810 186 (1592)</td>
</tr>
<tr>
<td>RazerS2</td>
<td>881 586</td>
<td>34 387</td>
<td>887 947</td>
<td>23 575</td>
<td>903 569 (4)</td>
</tr>
</tbody>
</table>

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.

#### 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 comparable performance with the heuristic methods of other aligners. These results show that some aligners cannot reliably map reads with a large number of mismatches.

**Table 4. Statistics for finding the unique hits of 1 000 000 reads taken from the 51 bp dataset ERR000577 and 100 bp dataset ERR024201 allowing different numbers of mismatches**

<table>
<thead>
<tr>
<th>Program</th>
<th>2-mis</th>
<th>3-mis</th>
<th>4-mis</th>
<th>5-mis</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>No. of Hits</td>
<td>Time (s)</td>
<td>No. of Hits</td>
<td>Time (s)</td>
</tr>
<tr>
<td>BatMis</td>
<td>817 844</td>
<td>38</td>
<td>843 466</td>
<td>55</td>
</tr>
<tr>
<td>BWA</td>
<td>817 844</td>
<td>191</td>
<td>841 850 (467)</td>
<td>397</td>
</tr>
<tr>
<td>RazerS2</td>
<td>816 536 (119)</td>
<td>10 567</td>
<td>822 838 (107)</td>
<td>10 669</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Program</th>
<th>100 bp</th>
<th>2-mis</th>
<th>3-mis</th>
<th>4-mis</th>
<th>5-mis</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>No. of Hits</td>
<td>Time (s)</td>
<td>No. of Hits</td>
<td>Time (s)</td>
<td>No. of Hits</td>
</tr>
<tr>
<td>BatMis</td>
<td>881 586</td>
<td>44</td>
<td>892 698</td>
<td>96</td>
<td>903 577</td>
</tr>
<tr>
<td>BWA</td>
<td>817 844</td>
<td>191</td>
<td>887 122 (167)</td>
<td>484</td>
<td>810 186 (1592)</td>
</tr>
<tr>
<td>RazerS2</td>
<td>881 586</td>
<td>34 387</td>
<td>887 947</td>
<td>23 575</td>
<td>903 569 (4)</td>
</tr>
</tbody>
</table>

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.

#### 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 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.

**Funding:** This work was supported in part by the Ministry Of Education, Singapore’s AcRF Tier 2 funding R-252-000-444-112.

**Conflict of Interest:** none declared.

**REFERENCES**


