BIOINFORMATICS ORIGINAL PAPER Vol. 28 no. 16 2012, pages 2122–2128
doi:10.1093/bioinformatics/bts339

Sequence analysis Advance Access publication June 10, 2012

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 and [2]Computational Biology Lab, School of Computing, National University of Singapore, 21 Lower Kent Ridge Road, Singapore 119077, Singapore

Associate Editor: 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 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 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.

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 (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; Markljung *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 (Lin *et al.*, 2008; Li *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 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

[†]The authors wish it to be known that, in their opinion, the first two authors should be regarded as joint First Authors.
[*]To whom correspondence should be addressed.

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 (*B*asic *A*lignment *T*ool for *Mis*matches) 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 $d(R[1..|R|],T[i..i+|R|-1])\leq k$. This article is also interested in reporting the occurrences in the order of increasing mismatches, i.e. we report i before i' if $d(R[1..|R|],T[i..i+|R|-1])<d(R[1..|R|],T[i.'.i'+|R|-1])$.

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 Σ. 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 ε. The suffix array $\mathrm{SA}_T[1..n]$ of T is a permutation of $\{1,\ldots,n\}$ such that, for any $i<j$, the suffix starting at position $\mathrm{SA}_T[i]$ is lexicographically smaller than the suffix starting at position $\mathrm{SA}_T[j]$.

Let P be a string. Suppose $\mathrm{SA}_T[i]$ and $\mathrm{SA}_T[j]$ 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 $B_T[i]=T[\mathrm{SA}_T[i]-1]$ for $\mathrm{SA}_T[i]>1$; 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 yP 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 $\mathrm{SA}_T[p], p=i,\ldots,j$, 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 $n-\mathrm{SA}_{T^*}[p], p=i,\ldots,j$, in T.

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

$$B_T=atc\$aaatc,\ B_{T^*}=acca\$atta$$

and

$$\mathrm{SA}_T=(9,8,3,1,4,2,5,7,6), SA_{T^*}=(9,8,5,6,1,7,4,3,2).$$

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_R^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 $\bigcup_{k=0}^{K}H_R^k$.

We define $R_l=R[1..\lfloor|R|/2\rfloor]$ and $R_r=R[\lfloor|R|/2\rfloor+1..|R|]$ 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_l and R_r by recursion. Precisely, it computes $H_{R_l}^k$ and $H_{R_r}^k$ for a set of values of k not exceeding $\lfloor K/2\rfloor$. 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_l^k and H_r^k 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 $d(R'_l,R_l)\leq k/2$ or $d(R'_r,R_r)\leq k/2-1$.*
- **Case 2: k is odd.** *We have either $d(R'_l,R_l)\leq (k-1)/2$ or $d(R'_r,R_r)\leq (k-1)/2$.*

PROOF. When k is even, from the pigeon hole principle, we have the cases $d(R'_l,R_l)\leq k/2$ or $d(R'_l,R_l)>k/2$. In the first case, the proof is obvious. In the second case we have

$$d(R'_r,R_r)\leq k-d(R'_l,R_l)\leq k-k/2-1=k/2-1.$$

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 $\bigcup_{i=0}^{k/2}H_{R_l}^i \cup \bigcup_{i=0}^{k/2-1}H_{R_r}^i$ to obtain all k-mismatch patterns of R. When k is odd, we extend patterns in $\bigcup_{i=0}^{(k-1)/2}H_{R_l}^i \cup \bigcup_{i=0}^{(k-1)/2}H_{R_r}^i$ 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, S*Ext*(X,R,k) performs suffix extension of the strings x in X to form another set Y of strings $y=\alpha\cdot x$ 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 $|X|>1$, $PExt(X,R,k)=\bigcup_{x\in X}PExt(\{x\},R,k)$

- If $X=\{x\}$, $PExt(\{x\},R,k)=$
$$\begin{cases}\{x\}, \text{ if } |x|=|R| \text{ or } d(x,R[1..|x|])=k+1\\ \bigcup_{\substack{\sigma\in\Sigma,\\ x\cdot\sigma\in T}} PExt(\{x\cdot\sigma\},R,k), \text{ otherwise}\end{cases}$$
- If $|X|>1$, $SExt(X,R,k)=\bigcup_{x\in X} SExt(\{x\},R,k)$
- If $X=\{x\}$, $SExt(\{x\},R,k)=$
$$\begin{cases}\{x\}, \text{ if } |x|=|R| \text{ or } d(x,R[|R|-|x|+1..|R|])=k+1\\ \bigcup_{\substack{\sigma\in\Sigma,\\ \sigma\cdot x\in T}} SExt(\{\sigma\cdot x\},R,k), \text{ otherwise}\end{cases}$$

Given these two procedures, the k-mismatch patterns of R can be computed as follows. When k is even, we report $\{x\in \mathrm{PE}xt\left(\bigcup_{i=0}^{k/2} H_{R_l}^i,R,k\right)\cup \mathrm{SE}xt\left(\bigcup_{i=0}^{k/2-1} H_{R_r}^i,R,k\right) \mid |x|=|R|, d(x,R)=k\}$. When k is odd, we report $\{x\in \mathrm{PE}xt\left(\bigcup_{i=0}^{(k-1)/2} H_{R_l}^i,R,k\right)\cup \mathrm{SE}xt\left(\bigcup_{i=0}^{(k-1)/2} H_{R_r}^i,R,k\right) \mid |x|=|R|, d(x,R)=k\}$.

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 $K+1$ iterations. For the kth iteration where $k=0,1,\dots,K$, our procedure tries to obtain H_R^k, i.e. all k-mismatch patterns of R. In the $0th$ iteration, we set $H_R^0=\{R\}$ if R exists in T; and $H_R^0=\emptyset$ otherwise. For the remaining iterations, we will not generate H_R^k starting from scratch. Instead, our routine will check all the unsuccessfully extended patterns from the $(k-1)^{st}$ iteration and see if they can be extended and become a k-mismatch pattern of R. Precisely, the kth iteration is divided into two stages. The first stage tries to extend those unsuccessfully extended patterns from the $(k-1)^{st}$ 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 $(k-1)^{st}$ iteration. By Lemma 2.2, if $k-1$ is odd, we need to extend the patterns in $\bigcup_{i=0}^{(k-2)/2} H_{R_l}^i \cup \bigcup_{i=0}^{(k-2)/2} H_{R_r}^i$ to obtain all the $(k-1)$-mismatch patterns of R. If $k-1$ is even, we need to extend the patterns in $\bigcup_{i=0}^{(k-1)/2} H_{R_l}^i \cup \bigcup_{i=0}^{(k-3)/2} H_{R_r}^i$ to obtain all the $(k-1)$-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 PRE_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 $\lceil k/2\rceil-1=(k-2)/2$ if $k-1$ is odd and $(k-1)/2$ if $k-1$ is even. In addition, $\lfloor k/2\rfloor-1=(k-2)/2$ 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 $d(x,R[1..|x|])=k$, $x[|x|]\neq R[|x|]$ and $d(x[1..|R_l|],R_l)\leq\lceil k/2\rceil-1$.
- Let SUF_R^k be the set of substrings x in T such that $d(x,R[|R|-|x|+1..|R|])=k$, $x[1]\neq R[|R|-|x|+1]$ and $d(x[|x|-|R_r|+1..|x|],R_r)\leq\lfloor k/2\rfloor-1$.

Intuitively, PRE_R^k contains a subset of the shortest substrings of T having exactly k mismatches with a prefix of R and SUF_R^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_R^1, PRE_R^2, and SUF_R^2.

LEMMA 2.3. *Consider a read R. Let* $P=\mathrm{PE}xt(H_{R_l}^0,R,1)$ *and* $S=\mathrm{SE}xt(H_{R_r}^0\cup SUF_R^1,R,1)$.

a) $H_R^1=\{x\in P\cup S \mid d(x,R)=1\}$.

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

c) $\mathrm{SUF}_R^2=\{x\in S \mid d(x,R[|R|-|x|+1..|R|])=2\}$.

PROOF. By Lemma 2.2, for any string R' which has 1 mismatch with R, we have either $R_l'=R_l$ or $R_r'=R_r$. Hence, H_R^1 contains all strings in $P\cup S$ whose patterns have exactly 1 mismatch with R. The equations for PRE_R^2 and SUF_R^2 follow by definition. □

```
$PExt(X,R,k)$
Require: A set $X$ of substrings in $T$ of length at most $|R|$ and
    have less than $k$ mismatches with a prefix of $R$.
Ensure: A set $\{z \mid x\in X, z=x\cdot y\in T, |z|\leq |R|, d(z,R[1..|z|])=k+1\}\cup\{z \mid x\in X, z=x\cdot y\in T, |z|=|R|, d(z,R)\leq k\}$.
Set $S=\emptyset$;
if $|X|>1$ then
    for each $x\in X$ do
        $S=S\cup PExt(\{x\},R,k)$;
    end for
else
    Let $x$ be the string in $X$;
    if $d(x,R[1..|x|])\leq k$ and $|x|<|R|$ then
        for $x\cdot\sigma\in T$ where $\sigma\in\{a,c,g,t\}$ do
            $S=S\cup PExt(\{x\cdot\sigma\},R,k)$;
        end for
    else
        $S=S\cup\{x\}$;
    end if
end if
Return $S$
```

```
$SExt(X,R,k)$
Require: A set $X$ of substrings in $T$ of length at most $|R|$ and
    have less than $k$ mismatches with a suffix of $R$.
Ensure: A set $\{z \mid x\in X, z=y\cdot x\in T, |z|\leq |R|, d(z,R[|R|-|z|+1..|R|])=k+1\}\cup\{z \mid x\in X, z=y\cdot x\in T, |z|=|R|, d(z,R)\leq k\}$.
Set $S=\emptyset$;
if $|X|>1$ then
    for each $x\in X$ do
        $S=S\cup SExt(\{x\},R,k)$;
    end for
else
    Let $x$ be the string in $X$;
    if $d(x,R[|R|-|x|+1..|R|])\leq k$ and $|x|<|R|$ then
        for $\sigma\cdot x\in T$ where $\sigma\in\{a,c,g,t\}$ do
            $S=S\cup SExt(\{\sigma\cdot x\},R,k)$;
        end for
    else
        $S=S\cup\{x\}$;
    end if
end if
Return $S$
```

Fig. 1. The procedures *PExt* and *SExt* perform prefix and suffix extensions, respective, of a set of strings X.

The following two lemmas state the recursive formulas to compute H_R^k, PRE_R^{k+1} and SUF_R^{k+1} for $k\geq 2$.

LEMMA 2.4. *Consider a read R and suppose k is odd. Let* $P=PExt(PRE_R^k,R,k)$ *and* $S=SExt(H_{R_r}^{\lfloor k/2\rfloor}\cup SUF_R^k,R,k)$.

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

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

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

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) $d(R'_l,R_l) \le (k-1)/2 = \lceil k/2 \rceil - 1$ (i.e. $R'_l \in \bigcup_{i=0}^{\lceil k/2 \rceil -1} H^i_{R_l}$) or (2) $d(R'_r,R_r) \le (k-1)/2 = \lfloor k/2 \rfloor$ (i.e. $R'_r \in \bigcup_{i=0}^{\lfloor k/2 \rfloor} H^i_{R_r}$). If R' satisfies (1), there should be some λ, where $|R'_l| \le \lambda \le |R'|$, such that $d(R'[1..\lambda],R[1..\lambda])=k$, with $R'[\lambda] \ne R[\lambda]$. 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, PRE^{k+1}_R equals

$$\begin{aligned} & \{x \in PExt(\cup_{i=0}^{\lceil (k+1)/2 \rceil -1} H^i_{R_l}, R, k) \mid d(x,R[1..|x|])=k+1\} \\ = \; & \{x \in PExt(\cup_{i=0}^{\lceil k/2 \rceil -1} H^i_{R_l}, R, k) \mid d(x,R[1..|x|])=k+1\} \\ = \; & \{x \in P \mid d(x,R[1..|x|])=k+1\} \end{aligned}$$

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

Lemma 2.5. *Consider a read R and suppose k is even. Let* $P = PExt(H^{\lfloor k/2 \rfloor}_{R_l} \cup PRE^k_R, R, k)$ *and* $S = SExt(SUF^k_R, R, k)$.

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

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

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

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 $0 \le k \le K$. The first phase (lines $1-2$) divides R into two equal halves R_l and R_r and it recursively calls *BatMis*$(R_l, \lfloor K/2 \rfloor)$ and *BatMis*$(R_r, \lfloor (K-1)/2 \rfloor)$ to compute $H^k_{R_l}$ and $H^k_{R_r}$ for $0 \le k \le \lfloor K/2 \rfloor$ and $0 \le k \le \lfloor (K-1)/2 \rfloor$, respectively. The second phase iteratively computes H^k_R for $k=0,\ldots,K$. It first computes H^0_R (line 3), H^1_R, PRE^2_R and PRE^2_R by Lemma 2.3 (lines $5-7$). Then, the program applies Lemmas 2.4 and 2.5 to compute $(H^k_R, \mathrm{PRE}^k_R, \mathrm{SUF}^k_R)$ iteratively for $k=2,\ldots,K$ (see lines $8-17$). Finally, the program reports H^k_R for $k=0,\ldots,K$.

```
BatMis(R, K)
Ensure: Report $H^k_R$ for $k = 0,1,\ldots,K$
    {Phase 1: Compute $H^k_{R_l}, H^k_{R_r}$ for $k = 0,1,\ldots,\lfloor K/2 \rfloor$}
 1: Compute $\{H^k_{R_l} \mid k = 0,1,\ldots,\lfloor K/2 \rfloor\}$ by calling
    $BatMis(R_l, \lfloor K/2 \rfloor)$;
 2: Compute $\{H^k_{R_r} \mid k = 0,1,\ldots,\lfloor (K-1)/2 \rfloor\}$ by calling
    $BatMis(R_r, \lfloor (K-1)/2 \rfloor)$;
    {Phase 2: Compute $H^k_R$ for $k = 0,\ldots,K$}
 3: Set $H^0_R = \{R\}$ if $R$ in $T$ and $\emptyset$ otherwise;
 4: for $k = 1$ to $K$ do
 5:     if $k = 1$ then
 6:         $P = PExt(H^0_{R_l}, R, 1)$;
 7:         $S = SExt(H^0_{R_r}, R, 1)$;
 8:     else if $k$ is odd then
 9:         $P = PExt(PRE^k_R, R, k)$;
10:         $S = SExt(H^{\lfloor k/2 \rfloor}_{R_r} \cup SUF^k_R, R, k)$;
11:     else
12:         $P = PExt(H^{\lfloor k/2 \rfloor}_{R_l} \cup PRE^k_R, R, k)$;
13:         $S = SExt(SUF^k_R, R, k)$;
14:     end if
15:     $H^k_R = \{x \in P \cup S \mid d(x,R) = k, |x| = |R|\}$.
16:     $PRE^{k+1}_R = \{x \in P \mid d(x, R[1..|x|]) = k+1\}$.
17:     $SUF^{k+1}_R = \{x \in S \mid d(x, R[|R|-|x|+1..|R|]) =$
        $k+1\}$.
18: end for
19: Return $(H^0_R, H^1_R, \ldots, H^K_R)$
```

Fig. 2. The BatMis algorithm.

2.5 Details of implementation

We need to find all occurrences of strings corresponding to $H^i_{R_l}, H^i_{R_r}, \mathrm{PRE}^j_R$ and SUF^j_R for $0 \le i \le \lfloor K/2 \rfloor, 0 \le j \le K$. Furthermore, we need an efficient way to extend them. To solve this problem, we build two BWT indexes B_T and B_{T^*}. All strings in $H^i_{R_l}$ and PRE^j_R are represented as SA_{T^*}-ranges, while all strings in $H^i_{R_r}$ and SUF^j_R are represented by their SA_T-ranges, where $0 \le i \le \lfloor K/2 \rfloor, 0 \le j \le K$. The extension of SA_{T^*}-ranges are done using forward search. Similarly, the extension of SA_T-ranges is done using backward search. The strings in H^k_R will in general contain a mixture of SA_T and SA_{T^*}-ranges, and it might be necessary to convert SA_T ranges to SA_{T^*} ranges or vice versa. This conversion is done by taking the string corresponding to SA_T(or SA_{T^*}) 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/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 $\mathrm{SA}_T[i]$ for every i which is a multiple of some fixed length κ. To compute $\mathrm{SA}_T[i]$, the algorithm counts the number of steps s' needed to arrive at a sampled point $B_T[s]$ by inverting the Burrows–Wheeler transform starting at $B_T[i]$. Then, $\mathrm{SA}_T[i] = SA_T[s] - s'$ is computed. SA_{T^*} needs to be sampled as SA_{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-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_T ranges to SA_{T^*} 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 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 SUF^i_R and PRE^i_R requires a lot of memory. To reduce memory usage, for $k > 5$, the set of k-mismatch hits, H^k_R, is computed directly based on Lemma 2.2. Although this approach reduces the memory required to store SUF^i_R and PRE^i_R 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

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

51 bp	0-mis		1-mis		2-mis		3-mis		4-mis		5-mis	
Program	No. of Hits	Time (s)	No. of Hits	Time (s)	No. of Hits	Time (s)	No. of Hits	Time (s)	No. of Hits	Time (s)	No. of Hits	Time (s)
BatMis	639 252	11	833 631	19	905 121	31	940 232	44	960 246	101	972 729	153
BWA	639 252	43	833 631	64	905 121	179	940 232	458	960 246	1979	972 726	9183
ZOOM	639 252	1007	833 631	1152	905 121	1731	**940 233(1)**	3182	**960 247(1)**	7699	–	–
RazerS2	639 252	16240	833 631	17 083	905 121	16 845	940 232	14 296	960 246	23 693	972 729	60 347

100 bp	2-mis		3-mis		4-mis		5-mis		8-mis		10-mis	
Program	No. of Hits	Time (s)	No. of Hits	Time (s)	No. of Hits	Time (s)	No. of Hits	Time (s)	No. of Hits	Time (s)	No. of Hits	Time (s)
BatMis	908 432	32	920 354	37	928 493	60	935 087	81	951 483	619	960 707	1003
BWA	908 432	193	920 354	257	928 493	574	935 087	2130	942 562	460 72	930 632	364 24
ZOOM	908 432	1612	920 354	2107	928 493	2418	–	–	–	–	–	–
RazerS2	908 432	340 04	920 354	328 92	928 493	327 87	935 087	25 692	951 483	663 05	960 707	121 884

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.

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 datasets 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 for handling a large number of mismatches. All the experiments were done on a Linux server running on 2×6-Core Intel Xeon X5680 Processors (3.33 Ghz), with 144 GB RAM. Each aligner was run in a single core, and the user time was reported.

All tested programs were run in their default settings with appropriate options for ignoring indels and enabling exact mapping set as needed. The demo version of ZOOM is only fully sensitive up to four mismatches. Because of this limitation, ZOOM was run allowing up to four mismatches only. Both BatMis and RazerS2 can report least mismatch hits and unique hits. For BWA, the default setting will report a unique hit if it exists or will otherwise report a random least mismatch hit. For ZOOM, the default mode outputs only the unique hits; it was made to output least mismatch hits with the -mk option. BWA, ZOOM and BatMis used the same reference genome where uncalled bases in the genome were replaced with a random nucleotide. For BWA and BatMis, the total time for the alignment and decoding the output is reported. For BatMis, post-processing was done to recover hits with boundary errors, and this time was added to the total time reported. All the command line parameters to obtain each table are described in the Supplementary material.

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 1 and 2.

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

51 bp	0-mis		1-mis		2-mis		3-mis		4-mis		5-mis	
Program	No. of Hits	Time (s)	No. of Hits	Time (s)	No. of Hits	Time (s)	No. of Hits	Time (s)	No. of Hits	Time (s)	No. of Hits	Time (s)
BatMis	586 314	13	759 079	21	817 844	38	843 466	55	856 383	125	863 707	191
BWA	586 314	43	759 079	64	817 844	179	843 466	458	856 383	1979	863 705	9183
ZOOM	586 314	1069	759 079	1190	817 844	1764	**843 467(1)**	3133	**856 384(1)**	7246	–	–
RazerS2	586 314	15 689	759 079	16 308	817 844	16562	**843 468(2)**	14 058	856 383	21 846	863 707	60 124
100 bp	2-mis		3-mis		4-mis		5-mis		8-mis		10-mis	
Program	No. of Hits	Time (s)	No. of Hits	Time (s)	No. of Hits	Time (s)	No. of Hits	Time (s)	No. of Hits	Time (s)	No. of Hits	Time (s)
BatMis	871 712	39	881 586	44	887 947	71	892 698	96	903 577	805	909 360	1389
BWA	871 712	193	881 586	257	887 947	574	892 698	2130	897 553	46 072	889 370	36 424
ZOOM	871 712	1597	881 586	2065	887 947	2311	–	–	–	–	–	–
RazerS2	871 712	31 580	881 586	30 195	**887 948(1)**	31 867	**892 701(3)**	26 616	**903 579(2)**	69029	**909 362(2)**	118 050

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

	1-mis		2-mis		3-mis		4-mis		5-mis	
Program	No. of Hits	Time	No. of Hits	Time	No. of Hits	Time	No. of Hits	Time	No. of Hits	Time
BatMis	12 709 391	85	31 001 496	208	59 121 695	412	96 502 481	1023	143 315 831	1803
BWA	**12 709 417(26)**	172	**31 001 544(48)**	411	**59 121 789(94)**	2601	**96 501 738(124)**	20 390	**143 186 148(168)**	163 479
ZOOM	**12 709 421(30)**	1581	**31 001 568(72)**	1703	**59 121 802(107)**	2307	**96 502 628(147)**	2771	–	–
RazerS2	12 709 391	30 778	31 000 921	31 956	591 19 338	32 340	96 498 204	32 747	143 309 572	26 770

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.

3.3 Multiple mappings

We benchmarked the time taken by each aligner to produce multiple mappings of a given read. We used the 100 bp real life dataset that were 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 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.

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

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

51 bp	2-mis		3-mis		4-mis		5-mis	
Program	No. of Hits	Time (s)	No. of Hits	Time (s)	No. of Hits	Time (s)	No. of Hits	Time (s)
BatMis	817 844	38	843 466	55	856 383	125	863 707	191
BWA	817 844	191	**841 850 (467)**	397	**852 875 (1400)**	438	**858 635 (2540)**	445
RazerS2	**816 536 (119)**	10 567	**842 838 (107)**	10 669	**855 989 (185)**	11 817	**863 604 (93)**	16 049
100 bp	2-mis		3-mis		4-mis		5-mis	
Program	No. of Hits	Time (s)	No. of Hits	Time (s)	No. of Hits	Time (s)	No. of Hits	Time (s)
BatMis	881 586	44	892 698	96	903 577	805	909 360	1389
BWA	**881 207 (48)**	257	**887 122 (167)**	484	**810 186 (1592)**	764	**904 356 (2782)**	922
Razers2	881 586	34 387	887 947	23 575	**903 569 (4)**	26 244	**909 347 (22)**	36 688

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.

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 100 bp reads, BWA misses between 2 and 9% hits, and for 51 bp reads it will miss 11 and 30% hits when there are 3–5 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.

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

Altschul,S. *et al.* (1990) Basic local alignment search tool. *J. Mol. Biol.*, **215**, 403–410.

Burrows,M. and Wheeler,D. (1994) A block-sorting lossless data compression algorithm. *Technical report*, Digital System Research Center.

Cox,A. (2006) Eland: efficient local alignment of nucleotide data.

Eckerle,L. *et al.* (2010) Infidelity of sars-cov nsp14-exonuclease mutant virus replication is revealed by complete genome sequencing. *PLoS pathogens*, **6**, e1000896–.

Ferragina,P. and Manzini,G. (2000) Opportunistic data structures with applications. *Technical report*.

Hillier,L. *et al.* (2008) Whole-genome sequencing and variant discovery in *c. elegans*. *Nat. Methods*, **5**, 183–188.

Hon,W. *et al.* (2004) Practical aspects of compressed suffix arrays and fm-index in searching dna sequences. In *ALENEX/ANALC*, pp. 31–38.

Lam,T. *et al.* (2008) Compressed indexing and local alignment of dna. *Bioinformatics*, **24**, 791–797.

Langmead,B. *et al.* (2009) Ultrafast and memory-efficient alignment of short dna sequences to the human genome. *Genome Biol.*, **10**, R25.

Lin,H. *et al.* (2008) Zoom! zillions of oligos mapped. *Bioinformatics*, **24**, 2431–2437.

Li,H. and Durbin,R. (2010) Fast and accurate long-read alignment with burrows-wheeler transform. *Bioinformatics (Oxford, England)*, **26**, 589–595.

Li,H. *et al.* (2008) Mapping short dna sequencing reads and calling variants using mapping quality scores. *Genome Res.*, **18**, 1851–1858.

Manber,U. and Myers,G. (1990) Suffix arrays: a new method for on-line string searches. In *Proceedings of the first annual ACM-SIAM symposium on Discrete algorithms*, Society for Industrial and Applied Mathematics, pp. 319–327.

Markljung,E. *et al*. (2009) Zbed6, a novel transcription factor derived from a domesticated dna transposon regulates igf2 expression and muscle growth. *PLoS Biol.*, **7**, e1000256–.

Mikkelsen,T. *et al.* (2007) Genome-wide maps of chromatin state in pluripotent and lineage-committed cells. *Nature*, **448**, 553–560.

Mortazavi,A. *et al.* (2008) Mapping and quantifying mammalian transcriptomes by rna-seq. *Nat. Methods*, **5**, 621–628.

Trapnell,C. *et al.* (2009) Tophat: discovering splice junctions with rna-seq. *Bioinformatics*, **25**, 1105–1111.

Weese,D. *et al.* (2009) Razersfast read mapping with sensitivity control. *Genome Research*, **19**, 1646–1654.