Reptile: representative tiling for short read error correction

Xiao Yang\textsuperscript{1,2}, Karin S. Dorman\textsuperscript{2,3,4} and Srinivas Aluru\textsuperscript{1,2,4,5,}\textsuperscript{*}

\textsuperscript{1}Department of Electrical and Computer Engineering, \textsuperscript{2}Bioinformatics and Computational Biology Program, \textsuperscript{3}Department of Statistics, \textsuperscript{4}Department of Genetics, Development & Cell Biology, Iowa State University, Ames IA 50011, USA and \textsuperscript{5}Department of Computer Science and Engineering, Indian Institute of Technology Bombay, Mumbai 400076, India

ABSTRACT

\textbf{Motivation:} Error correction is critical to the success of next-generation sequencing applications, such as resequencing and \textit{de novo} genome sequencing. It is especially important for high-throughput short-read sequencing, where reads are much shorter and more abundant, and errors more frequent than in traditional Sanger sequencing. Processing massive numbers of short reads with existing error correction methods is both compute and memory intensive, yet the results are far from satisfactory when applied to real datasets.

\textbf{Results:} We present a novel approach, termed Reptile, for error correction in short-read data from next-generation sequencing. Reptile works with the spectrum of \textit{k}-mers from the input reads, and corrects errors by simultaneously examining: (i) Hamming distance-based correction possibilities for potentially erroneous \textit{k}-mers; and (ii) neighboring \textit{k}-mers from the same read for correct contextual information. By not needing to store input data, Reptile has the favorable property that it can handle data that does not fit in main memory. In addition to sequence data, Reptile can make use of available quality score information. Our experiments show that Reptile outperforms previous methods in the percentage of errors removed from the data and the accuracy in true base assignment. In addition, a significant reduction in run time and memory usage have been achieved compared with previous methods, making it more practical for short-read error correction when sampling larger genomes.

\textbf{Availability:} Reptile is implemented in C++ and is available through the link: http://aluru-sun.ece.iastate.edu/doku.php?id=software

\textbf{Contact:} aluru@iastate.edu

Received on March 16, 2010; revised on August 9, 2010; accepted on August 10, 2010

1 INTRODUCTION

High-throughput sequencing is profoundly changing the way genetics data are collected, stored and processed (Shendure and Ji, 2008). The advantages of the new technology have led to revitalization of old techniques and discovery of novel uses, with growing applications in resequencing, \textit{de novo} genome assembly, metagenomics and beyond (Ansorge, 2009; Marguerat et al., 2009). New technology inevitably comes with challenges. For many next-generation sequencers, the advantage of deeper and cheaper coverage comes at the cost of shorter reads with higher error rates compared with the Sanger sequencing they replace.

Genome assembly, the \textit{de novo} inference of a genome without the aid of a reference genome, is challenging. Sanger reads, typically 700–1000 bp in length, are long enough for overlaps to be reliable indicators of genomic co-location, which are used in the overlap-layout-consensus approach for genome assembly. However, this approach does poorly with the much shorter reads of many next-generation sequencing platforms (e.g. 35–100 bp for Illumina Genome Analyzer II). In this context, de Bruijn graph (Idury and Waterman, 1995) and string graph (Myers, 2005) based formulations that reconstruct the genome as a path in a graph perform better due to their more global analysis and ability to naturally accommodate paired read information. As a result, they have become \textit{de facto} models for building short-read genome assemblers, e.g. ALLPATHS (Butler et al., 2008), Velvet (Zerbino and Birney, 2008), ABySS (Simpson et al., 2009) and Yaga (Jackson et al., 2010). Error correction has long been recognized as a critical and difficult part of these graph-based assemblers. It also has significant impact in other next-generation sequencing applications such as resequencing.

We give a brief review of several well-known error correction methods. Alignment-based error correction methods, such as MistEd (Tammi et al., 2003) for Sanger reads, require refined multiple read alignments and assume unusually isolated bases to be read errors. Like the Sanger-motivated assembly algorithms, these approaches do not adapt well to short reads. Hence, Chaisson et al. (2004) proposed the spectral alignment problem (SAP): in a given dataset, a \textit{k}-mer is considered \textit{solid} if its multiplicity exceeds a threshold, and \textit{insolid} otherwise. Reads containing \textit{insolid} \textit{k}-mers are corrected using a minimum number of edit operations so that they contain only \textit{solid} \textit{k}-mers post-correction. Similar approaches have been adapted and used by others (Butler et al., 2008). To overcome the typically long run times of SAP-based approaches, Schröder et al. (2009) proposed SHREC, a method based on a generalized suffix tree constructed from short-read data using both forward and reverse complementary strands. SHREC compares the multiplicity of a substring, represented by a node in the suffix tree, with its expected frequency of occurrence calculated analytically, assuming uniform sampling of the genome and uniformly distributed sequencing errors. The nodes with observed counts that deviate beyond a tolerable threshold from their expected values are considered erroneous. An erroneous node is corrected to a sibling when applicable, and all its descendants are transferred to the selected sibling. Well-engineered code is necessary to cope with the large
memory requirement of the suffix tree data structure. Unlike these general purpose error correction methods, FreClu (Qu et al., 2009) targets transcripome data. The error rates for each position of a read are estimated in the same experiment via a set of control reads of a known bacterial artificial chromosome (BAC) sequence. Reads are clustered using the estimated error rates, and after error correction, FreClu could map $\sim$5% more reads back to the reference genome.

Error correction of short-read data is particularly challenging because of the massive datasets, non-uniformly distributed read errors introduced at relatively high rates, and non-uniform coverage of the target genome. Next-generation short-read sequencers produce hundreds of millions of reads in a single run, and this trend of fast, massive data generation is continuing to accelerate. To process these data, even an efficient linear space algorithm could easily exceed available memory on a standard desktop computer. The high rate of sequencing errors also significantly increases memory usage due to the introduction of numerous erroneous kmers that are not present in the genome. Errors are typically identified as the unusual reads or kmers occurring less frequently than a threshold calculated under the assumptions of uniform coverage and error distribution. Neither assumption holds true in real data, leading to an excess of false error predictions.

In this article, we present Reptile, a scalable short-read error correction method that effectively addresses the above challenges. We draw upon the k-spectrum approach pioneered in earlier results, but explore multiple alternative kmer decompositions of an erroneous read and use contextual information specified by neighboring kmers to infer appropriate corrections. Reptile also incorporates quality score information when available. We present algorithmic strategies to store kmer Hamming distance graphs and efficiently retrieve all graph neighbors of a kmer as candidates for correction. We compare our results with SHREC (Schröder et al., 2009), a high-quality short-read error correction method, and one of the more recent in a line of continuously improving error correction methods that effectively address the above challenges. For instance, base Т in $r_1$ would be considered a sequencing error to be corrected to the consensus base A.

3 METHODS

The success of any error correction method relies on an adequate coverage of the target genome. If we know the genomic location of every read, we could layout all reads that contain a specific genomic position into a multiple alignment (Fig. 1) and correct all erroneous bases to the consensus base under the reasonable assumption that errors are infrequent and independent. For instance, base Т in $r_1$ would be considered a sequencing error to be corrected to the consensus base A.

The main idea underlying Reptile is to create approximate multiple alignments, with the possibility of substitutions, in the absence of location information. Multiple alignments with substitutions could be created by considering all reads with pairwise Hamming distance less than some threshold, but such alignments are already hard (Manthey and Reischuk, 2005) and even in high coverage situations, the occurrence of many exactly coincident reads, e.g. $r_1$ and $r_2$ in Figure 1, are rare. We therefore resort to alignments on subreads, the substrings of a read. Storing $R$, let alone all its subreads, could be memory intensive, not to mention the memory required to store information required for error correction. Inspired by the idea for bounding memory usage with de Bruijn graphs in short-read assembly, we work with kmer subreads of input data, where the memory of storing the k-spectrum $R^k$ is bounded by $O(min(4^k, L(\log k + 1)))$. Typically, $k$ is chosen so that the expected number of occurrences of any kmer in the genome should be no more than one, i.e. $4^k > |G|$. Therefore, choosing $10 \leq k \leq 16$ is sufficient for microbial or human genomes, in which case the k-spectrum would fit within 4 GB RAM regardless of input size.

Focusing on reasonably short kmers has several advantages. First, we expect an adequate number of kmers to align to the same position along the genome even with relatively low coverage (e.g. 40×). High local coverage is needed to identify erroneous bases. For instance, in Figure 1, there exist five subreads, four copies of $a_2$ and one copy of $a'_2$, aligning to the same starting position in the genome, but this number reduces to three for the longer subread $a_2|\sigma_3$. Second, it is less compute intensive to identify $N^d$ when $k$ is small, since there are fewer ways to select $d$ out of $k$ positions. Last, sequencing errors in kmers are much less frequent compared with full-length reads, so $d$ need not be large.

Let $R = \{r_1, r_2, \ldots, r_l\}$ be a collection of short reads sequenced from genome $G$. For simplicity (but without loss of generality), we assume each read $r_i$ has a fixed length $L$. The coverage of the genome by the reads is given by $Cov = \sum_{i=1}^{l} r_i$, where $|G|$ denotes the genome length.

Define the k-spectrum of a read $r$ to be the set $R^k = \{r[i:i+k-1] | 0 \leq i \leq L-k+1\}$, where $r[i:j]$ denotes the substring from position $i$ to $j$ in $r$. We index the positions of a string starting from 0. The k-spectrum of $R$ is given by $R^k = \bigcup_{i=1}^{l} R^k_i$.

Let $a$ and $b$ be two strings such that $a \equiv (|a| \equiv 1); (|a| \equiv 1)$ is $\emptyset$. We define the $l$-concatenation of $a$ and $b$, denoted $a[b]$, to be the string $\gamma$ of length $|a| + |b| - l$ such that $\gamma(0:|a|) = \gamma(|a|) = \gamma(|a| + |b| - l)$. The Hamming distance between two strings $a_1$ and $a_2$, denoted $hd(a_1, a_2)$, is the number of positions at which they differ. For a kmer $a_1 \in R^k$, define its d-neighborhood $N^d_{a_1} = \{a_2 \in R^k | \text{hd}(a_1, a_2) \leq d\}$. Its complete d-neighborhood $N^{\infty}_{a_1} = \{a_2 | \text{hd}(a_1, a_2) \leq d\}$ contains all kmers within Hamming distance $d$, whether or not they occur in $R^k$.

2 NOTATIONS

Let $R = \{r_1, r_2, \ldots, r_l\}$ be a collection of short reads sequenced from genome $G$. For simplicity (but without loss of generality), we assume each read $r_i$ has a fixed length $L$. The coverage of the genome by the reads is given by $Cov = \sum_{i=1}^{l} r_i$, where $|G|$ denotes the genome length.

Define the k-spectrum of a read $r$ to be the set $R^k = \{r[i:i+k-1] | 0 \leq i \leq L-k+1\}$, where $r[i:j]$ denotes the substring from position $i$ to $j$ in $r$. We index the positions of a string starting from 0. The k-spectrum of $R$ is given by $R^k = \bigcup_{i=1}^{l} R^k_i$.

Let $a$ and $b$ be two strings such that $a \equiv (|a| \equiv 1); (|a| \equiv 1)$ is $\emptyset$. We define the $l$-concatenation of $a$ and $b$, denoted $a[b]$, to be the string $\gamma$ of length $|a| + |b| - l$ such that $\gamma(0:|a|) = \gamma(|a|) = \gamma(|a| + |b| - l)$. The Hamming distance between two strings $a_1$ and $a_2$, denoted $hd(a_1, a_2)$, is the number of positions at which they differ. For a kmer $a_1 \in R^k$, define its d-neighborhood $N^d_{a_1} = \{a_2 \in R^k | \text{hd}(a_1, a_2) \leq d\}$. Its complete d-neighborhood $N^{\infty}_{a_1} = \{a_2 | \text{hd}(a_1, a_2) \leq d\}$ contains all kmers within Hamming distance $d$, whether or not they occur in $R^k$.

3 METHODS

The success of any error correction method relies on an adequate coverage of the target genome. If we know the genomic location of every read, we could layout all reads that contain a specific genomic position into a multiple alignment (Fig. 1) and correct all erroneous bases to the consensus base under the reasonable assumption that errors are infrequent and independent. For instance, base Т in $r_1$ would be considered a sequencing error to be corrected to the consensus base A.

The main idea underlying Reptile is to create approximate multiple alignments, with the possibility of substitutions, in the absence of location information. Multiple alignments with substitutions could be created by considering all reads with pairwise Hamming distance less than some threshold, but such alignments are already hard (Manthey and Reischuk, 2005) and even in high coverage situations, the occurrence of many exactly coincident reads, e.g. $r_1$ and $r_2$ in Figure 1, are rare. We therefore resort to alignments on subreads, the substrings of a read.

Storing $R$, let alone all its subreads, could be memory intensive, not to mention the memory required to store information required for error correction. Inspired by the idea for bounding memory usage with de Bruijn graphs in short-read assembly, we work with kmer subreads of input data, where the memory of storing the k-spectrum $R^k$ is bounded by $O(min(4^k, L(\log k + 1)))$. Typically, $k$ is chosen so that the expected number of occurrences of any kmer in the genome should be no more than one, i.e. $4^k > |G|$. Therefore, choosing $10 \leq k \leq 16$ is sufficient for microbial or human genomes, in which case the k-spectrum would fit within 4 GB RAM regardless of input size.

Focusing on reasonably short kmers has several advantages. First, we expect an adequate number of kmers to align to the same position along the genome even with relatively low coverage (e.g. 40×). High local coverage is needed to identify erroneous bases. For instance, in Figure 1, there exist five subreads, four copies of $a_2$ and one copy of $a'_2$, aligning to the same starting position in the genome, but this number reduces to three for the longer subread $a_2|\sigma_3$. Second, it is less compute intensive to identify $N^d$ when $k$ is small, since there are fewer ways to select $d$ out of $k$ positions. Last, sequencing errors in kmers are much less frequent compared with full-length reads, so $d$ need not be large.

Fig. 1. $G$ is the target genome, shown as a bold line; $r_1$′s (0 ≤ j ≤ 8) represent reads, shown as thin lines; $a_1$ (0 ≤ j ≤ 8), $a'_2$, and $a''_2$ are kmer instances in the reads, shown as rectangles. Every read is drawn aligned to its origin of sequencing position on the target genome. The bases at two positions in the kmers $a_2$, $a'_2$, and $a''_2$ are shown. All other positions in these kmers match across all three variants.
local, approximate alignments of tilings constructed from d
when resolving erroneous bases. For instance, in Figure 1, without knowing
the alignment, it is unclear if u2 should be corrected to x or x' since both
h(u2|u2)≈h(x|u2)≈1 and a2 and a2 have a similar higher frequency. However, the contextual information of u2 available from read r2 (in this
case, a2) uniquely identifies u2 as the right correction. We seek a way to
use contextual information to help resolve errors without increasing k and lowering local coverage.

Contextual information is provided by noting which contexts co-exist in
observed reads. For instance, u1 || u2 exists in r1 or r2, and u3 || u4 exists in r3 in Figure 1. The following definitions formalize the notion of
contextual information.

Definition 1. r = u1 || u2 (0 ≤ l < k) is a tile of read r if t is a substring of
v, and |u1| = |u2| = k.

Definition 2. t' = u'(l)j' is a d-mutant tile of t = u(l)j if h(u(l)j) ≤ d and
h(u'(l)j') ≤ d.

Definition 3. Tt = (τ1, τ2, . . . , τm) is a tiling of read r if τi = |t|1 ≤ i ≤ m is a tile of r, and (2) li ≥ 1 (1 ≤ i ≤ m).

Note that if τi is specified as r(l)j, then the overlaps between consecutive tiles can be inferred; i.e. k's can be derived from τi. Multiple
tilings exist for any read. For example, both ((τ1)||u(τ1)||u(τ1)||u(τ1)) and
((τ1)||u(τ1)||u(τ1)||u(τ1)) are tilings of r. We also extend the concept of d-mutant tiles to tilings. For instance, we can think of
(τ1)||u(τ1)||u(τ1)||u(τ1)) as a one-mutant tiling of Tτ = (τ1||u(τ1)||u(τ1)||u(τ1)). Similarly, (τ1)||u(τ1)||u(τ1)||u(τ1)) is a two-mutant tiling of Tτ. Formally:

Definition 4. A d-mutant tiling of Tτ = (τ1, τ2 . . . , τm) is a sequence of
tiles (τ′1, τ′2, . . . , τ′m) such that (1) li = li and (2) li = li for 1 ≤ i ≤ m, where
li is implicitly determined by r for 1 ≤ i ≤ m, and (2) li is a d-mutant tile of
τi for 1 ≤ i ≤ m.

If read r contains errors and Tτ is a tiling of r, then we expect to find a
tiling Tτ of the true read s as one of the d-mutant tilings of Tτ, where
consistent tiles of Tτ have higher coverage than those of Tτ. However, in
some cases, Tτ will not be found among the d-mutant tilings.

3.3 The algorithm
For ease of presentation, we assume R can fit in main memory (this
requirement will be relaxed later). The algorithm consists of two major
phases. We first provide a brief overview, and subsequently describe the
algorithm in more detail and analyze its time and space complexity.

(1) Information extraction.
(a) Derive the k-spectrum Rk of R.
(b) Derive Hamming graph Gp = (Vp, Ep), where v1 ∈ Vp represents
u1 ∈ Rk and edge e = (v1, v2) ∈ Ep ⇔ h(u1, u2) ≤ d, for a
given threshold d.
(c) Compute tile occurrences.
(2) Individual read error correction.
(a) Place an initial t at the beginning of the read.
(b) Identify d-mutant tiles of t.
(c) Correct errors in t as applicable.
(d) Adjust tile placement and go to step 2b, until tile placement
choices are exhausted.

Given a read r ∈ R, any of its constituent 1mers α is a vertex v in the
Hamming graph. The d-neighborhood of α is accessible via the edges
incident to v. Hence, if α contains at most d substitution sequencing errors, the
1mer it should be corrected to exists in its d-neighborhood. By building
local, approximate alignments of tilings constructed from d-neighbors,
our strategy amounts to aligning the true read as a high frequency tiling.

3.1.1 Phase 1: information extraction. Constructing Rk involves one
linear scan of each read in R. This takes O(nL) time. We maintain Rk in
sorted order using O(2(nL log Rk)) time. The space requirement for Rk is
given by |Rk| = O(min(4d, nL − k + 1)). Any non-ACGT characters (due to
difficulty in base calling) are initially converted to A, which will be validated
or corrected later by the algorithm.

During error correction, it is important to have fast access to the d-
neighborhood of any 1mer, ideally in constant time per neighboring 1mer.
One could do so by storing the entire Hamming graph Gp, but it would
require large amount of memory. If we assume G as a random string, and
errors accumulate independently with probability p, then the probability that
a node is linked to another is p2 = 256−d/3 · 0.75d + 1−p2−d/3, including the chance that another random 1mer in the genome is within d
Hamming distance of the current 1mer and the chance that the current 1mer
contains up to d errors. Thus, the expected memory usage is O(1.39k)|p|.

Alternatively, we could binary search considering only indices used for sorting Rk. All 1mers that belong to the d-neighborhood of α are adjacent to α, in at least
one Rk. If sequencing errors accumulate independently, then the expected
number of elements of Rk found in each Rk is |Rk|/4k−d/3−k. Hence,
we need approximately 1.39k|Rk|/4d−d/3 time to expect to recover all edges
of the Hamming graph, i.e. O(1.39k|Rk| log Rk) time assuming both k and h
are constants. Typically, |Rk| < 4h, therefore, choosing a larger v value
will use more memory, but less expected run time. As an example, in a
real Escherichia coli dataset with 160x coverage, storing 13 copies of Rk
required only ~560 MB memory, but the average number of hits per 1mer
in each 13-spectrum was less than one. Therefore, identifying each element
of Rk for a 13-mer took constant time on average.

The above method provides an exact solution for identifying all edges in
the Hamming graph. Alternatively, a simpler recursive approximation derives
N. by inferring N. for each element in N. This strategy might be more
biologically meaningful (Qu et al., 2009) but is only an approximation since
an edge between two vertices v and v could be recovered only if there exists a
path connecting them such that adjacent vertices represent 1mers that differ
by exactly one position. In this case, choosing a smaller k, using a larger
dataset, or having a higher sequencing error rate all improve the chance to
identify all edges.

Tiles are l-concatenations of consecutive or overlapping 1mers found in
reads. Here, we use one fixed value of l but different values of l can be
used to consider tiles with different lengths. We compute the multiplicities of
tiles by a linear scan of every read to record all tiles, followed by a sort
of collected tiles and one linear scan of the sorted list. This process takes
O(|Rk| log|Rk|) time, where |Rk| = O(min(4d, nL − k + 1)). Meanwhile, we record the number of occurrences of each tile, where every
position has a quality score exceeding some threshold Q. Typically, a quality
score is associated with every base of a short read. The score indicates the
probability p that the corresponding base is sequenced incorrectly.
For instance, Illumina GenomeAnalyzer encodes the quality score as $Q = -10\log_{10}(p/(1-p))$. A higher score indicates a more reliable base call.

To deal with the double strandedness of the target genome, we consider both the forward and reverse complementary strands of every read. Edge identification in the de Bruijn graph takes twice the time, but no additional memory is needed since $B^d$ is already generated using both strands.

3.1.2 Phase 2: error correction We use the contextual information in read $t$ to identify sequencing errors through the process of choosing a tiling $T_t$ and comparing it with $d$-mutant tilings. In particular, if $t$ contains $s$ errors, and we choose any tiling $T_t$, then an error-free tiling $T_t'$ belongs to the collection of $d$-mutant tilings of $T_t$ if $d \geq s$. Under the standard assumption of uniform coverage, the tiles of $T_t$ should be substantially more abundant than at least some of the tiles of $T_t'$ with errors. After $T_t$ is identified, the true read $t$ can be readily inferred from $T_t$.

In practice, $s$ could be large, and sequencing errors tend to cluster toward the 3' end of a read. Since we prefer $d$ to be small to limit memory usage, run time and false error detection, it is entirely possible that $T_t'$ is not one of the $d$-mutant tilings of $T_t$. On the other hand, an alternate tiling $G_t$ of $t$ may lower the maximum number of mutations per kmer to below $d$ such that $G_t$ with high frequency tilings is one of the $d$-mutant tilings of $G_t$. In the case that there is no such tiling $G_t$, we examine a subset of constituent tiles in $G_t$. If a high coverage path of these selected tiles is present, the tiles are corrected. With some errors removed, a tiling may now exist that contains the true read among its $d$-mutant tilings.

These observations are sufficient to motivate the following procedure for identifying and correcting read errors. Place a tile on $t$ and attempt to correct it via comparisons with its $d$-mutant tiles (tile correction). If it is validated or corrected, move to the next tile in the standard tiling and repeat. If it cannot be corrected or validated, look for an alternative tiling, presumably one that avoids clusters of more than $d$ errors that are thwarting attempts to find error-free tiles within the 5' neighbourhoods. We first describe tile correction in Algorithm 1, then the overall procedure for read correction in Algorithm 2.

Tile correction. For each tile in $R$, we have recorded its multiplicity $O_t$ in $R$ and the number $Q_t$ of those instances where the quality score of every base exceeds $Q$. If a short read dataset comes with unreliable or missing quality score information, we set $O_t = 0$, otherwise, $O_t$ is a better estimate of the number of error-free occurrences of each tile.

The tile correction procedure is given in Algorithm 1. A decision to correct tile $t$ is based on the high-quality occurrence counts $O_t$ of $t$ compared with its $d$-mutant tiles. As a rule of thumb, there must be compelling evidence before a correction is made. Any tile is automatically validated if it occurs count exceeds an upper threshold $C_t$ (lines 1–2). If there is no occurrence tile with no $d$-mutant tiles is validated only if it occurs more than a low threshold $C_t$ times (lines 4–6). When there exist $d$-mutant tilings of $t$ with much higher frequencies (at least $C_t$ times, $C_t > 1$) than $t$ and which differ at low-quality bases ($< Q$), it is likely that $t$ contains errors and the tile is one of its $d$-mutant tiles. In such cases, a correction is possible only if there is a unique $d$-mutant tile with the closest Hamming distance, including a mutation at a low-quality base in $t$ (line 14). We will replace $t$ with this tile. Otherwise, we choose not to modify $t$ to avoid false corrections.

Similarly, the same strategy is applied in $T_t$ with very low multiplicity (lines 17 and 18). Since there exist a constant number of $d$-mutant tiles of $t$, and correcting $t$ requires comparison of every base between 1 and $t' \in T_t$, Algorithm 1 takes constant time.

Read correction. The overall procedure for correcting a read is given in Algorithm 2. In line 1, an initial tile is chosen and $d_1$ and $d_2$, two parameters specifying the maximum Hamming distance allowed in the two constituent kmers while identifying mutant tiles, are initialized to $d$. The algorithm terminates when no more plausible tiles can be identified. Within the while loop, $d$-mutant tiles are identified for the current tile (line 3), which is validated or corrected using Algorithm 1 (line 4). The new placement of $t_{\text{new}}$, according to decisions, [D1] through [D3] (lines 6–8), gradually forms a validated or corrected tiling of read $t$, although some reads may never be fully validated.

To better understand how the rules in lines 6–8 choose a tiling, we illustrate with an example in Figure 2 for $d = 1$. An initial tile $t_1$ is chosen as shown. Since there exists one error in each of the constituent kmers, $t_1$ can be corrected. Tile $t_1$ is chosen as the next tile according to [D2], but two sequencing errors within the second inner of $t_1$ lead to an inconclusive decision. Hence, $t_1$ is not selected in the tiling and another alternative tile $t_1$ is chosen according to [D3(a)]. The algorithm iterates and if tiles can be validated or corrected at every stage, we are able to complete a tiling moving from 5' to 3' along the read. Unfortunately, non-uniform coverage and the existence of more than one reasonable $d$-mutant tile can lead to an inconclusive decision in Algorithm 1 regardless of tiling choice. In Figure 2, the 5’ to 3’ tiling encounters an inconclusive dead-end at arrow $d_1$. To move past dead-end tiles, a non-overlapping tiling $t_2$ is chosen according to [D3(b)]. A small unvalidated gap is left in the middle of the 5’ to 3’ tiling of this read. The example shows only the tiling from 5’ to 3’. The same strategy is applied in the 3’ to 5’ direction.

We briefly analyze the run time of Algorithm 2. Tiles are sorted, so tile information is accessed in $O(\log(nL))$ time. Therefore, line 3 requires $O(\log(nL))$ time. Once some tiles have been corrected, the search space for new $d$-mutant tiles shrinks when $d_1$ or $d_2$ is set to 0 in line 5. The maximum number of non-overlapping tiles in a tiling is the constant $L/\mu$. Hence, the time spent in correcting each read is $O(\log(nL))$, and the overall run time of Algorithm 2 is $O(t_1(\log(nL)))$.

3.1.3 Choosing parameters Although analytical calculations like those adopted in existing methods can be used to choose parameters of Algorithm 1, we choose their values based on the input data to help avoid the unrealistic assumptions of uniformly distributed read errors and uniform genome coverage. Given short-read data $R$, we examine the empirical distribution of quality scores and choose threshold $Q$ such that a given percentage (e.g. 15–20%) of bases have quality score value below $Q$. This value could

Algorithm 1

Tile $t$ error correction.

1. if $O(t) > C_t$ then
2. $t$ is valid; return.
3. end if
4. if $t$ has no $d$-mutant tiles $t'$ then
5. if $O(t') \geq C_m$ then
6. $t$ is valid; return.
7. end if
8. return due to insufficient evidence.
9. end if
10. if $O(t) \geq C_m$ then
11. Select $d$-mutant tiles $T_t = \{t' \mid O(t') \geq C_m\}$.
12. if $T = \emptyset$, $t$ is valid; return.
13. If $T = \emptyset$, $t$ is valid; return.
14. Correct $t$ to $t'$ and return if $(t \not\subseteq \Delta) \cup (t' \not\subseteq \Delta)$ for all $t' \in T$.
15. Correct $t$ to $t'$; return.
16. else
17. if $t$ has only one $d$-mutant tile $t'$ s.t. $O(t') \geq C_m$ then
18. Correct $t$ to $t'$; return.
19. else
20. return due to insufficient evidence.
21. end if
22. end if

Algorithm 2

Tile correction.

1. An initial tile $t_1$ is chosen.
2. If $t_1$ is a valid tiling, record $t_1$ and exit.
3. Choose a new tile $t_2$ according to [D1]. The algorithm iterates and if tiles can be validated or corrected at every stage, we are able to complete a tiling moving from 5' to 3' along the read. Unfortunately, non-uniform coverage and the existence of more than one reasonable $d$-mutant tile can lead to an inconclusive decision in Algorithm 1 regardless of tiling choice. In Figure 2, the 5’ to 3’ tiling encounters an inconclusive dead-end at arrow $d_1$. To move past dead-end tiles, a non-overlapping tiling $t_2$ is chosen according to [D3(b)]. A small unvalidated gap is left in the middle of the 5’ to 3’ tiling of this read. The example shows only the tiling from 5’ to 3’. The same strategy is applied in the 3’ to 5’ direction.

We briefly analyze the run time of Algorithm 2. Tiles are sorted, so tile information is accessed in $O(\log(nL))$ time. Therefore, line 3 requires $O(\log(nL))$ time. Once some tiles have been corrected, the search space for new $d$-mutant tiles shrinks when $d_1$ or $d_2$ is set to 0 in line 5. The maximum number of non-overlapping tiles in a tiling is the constant $L/\mu$. Hence, the time spent in correcting each read is $O(\log(nL))$, and the overall run time of Algorithm 2 is $O(t_1(\log(nL)))$. 

2529
Algorithm 2 Read error correction.

1. Initialize: \( r \leftarrow R_0 \), where \( R_0 \) is a prefix of \( r \); \( d_1, d_2 \leftarrow d \).
2. while \( r \neq \emptyset \) do
3. Identify \( d \)-mutant tiles of \( t = a_1 \ldots a_t \) as the set \( \{ t' = a'_1 \ldots a'_t | (a'_1, a'_2) \in N_1 \times N_2^d \} \).
4. Correct \( r \) (Algorithm 1).
5. Based on the decisions made on \( r \) in the former step, tile \( t_{ew} \) of \( r \) will be chosen according to \([D1] - [D3]\) as follows. If there is insufficient space to place a tile toward the end of a read, \( t_{ew} \) will be chosen as the suffix of \( r \).
6. \([D1] \) is valid: select \( t_{ew} \) such that the suffix–prefix overlap between \( r \) and \( t_{ew} \) equals \( d_1 - d_2 \leftarrow 0 \).
7. \([D2] \) was corrected to \( r' = a'_1 \ldots a'_t + 1 \) and \( t \) is replaced with \( r' \) in \( r \): select \( t_{ew} \) such that the suffix–prefix overlap between \( r \) and \( t_{ew} \) equals \( a'_1 - d_1 \leftarrow 0 \).
8. \([D3] \) Insufficient evidence to correct \( r \): select \( t_{ew} \) to be one of the following. Let \( t_{1} \ldots t_{2} \) be a maximal validated or corrected region of \( r \) that overlaps with the 5' region of \( r \), where \( t_2 \) starts with \( t_1 + 1 \geq |r| \), and \( t_1 \) and \( t_2 \) are a maximal uncorrected region from previous iterations due to insufficient evidence.
   a. if \( t_2 + 1 \ldots t_3 = \emptyset \), then \( t_{ew} \leftarrow r [-|t_2| - 2, t_2 + 1] \) and \( d_2 \leftarrow 1 \).
   b. if \( t_2 + 1 \ldots t_3 \neq \emptyset \), then \( t_{ew} \leftarrow r [t_1, t_1 + |t_1|] \).
   Similarly, if \( t_1 \ldots t_2 \) overlaps with \( Y \) end \(- \) of \( r \) and \( t_1 \ldots t_1 - 1 \) is a maximal uncorrected region from previous iterations, then if \( r[t_1, t_1 - 1] = \emptyset \), set \( t_{ew} \leftarrow r [-|t_2| - 2, t_2 + 1] \) and \( d_2 \leftarrow 1 \).
   In above cases, \( t_{ew} \) is valid only if it was not assigned with the same value in previous iterations (unless \( t_{ew} \) became the suffix of \( r \)). If such a choice does not exist, \( t_{ew} \leftarrow \emptyset \).
9. \( t \leftarrow t_{ew} \).
10. end while

Fig. 2. An illustration of choosing a tiling of a read for \( d = 1 \). A read is represented on top by a concatenation of rectangles, where each rectangle denotes a k-mer. Each tile is represented by a concatenation of two adjacent arrows, which denote its k-mer composition. For simplicity of illustration, we choose the read length to be divisible by \( k \) and each tile is a 0-1 concatenation of two adjacent k-mers. X's denote sequencing errors. Each bold arrow, \( a_i (1 \leq i \leq 4) \), denotes tile with insufficient evidence for correction. The placement of an alternative tile is indicated by a dotted arrow.

be adjusted to consider the error rates of the particular next-generation sequencing equipment in use. Given \( Q_i \) and counts of high-quality tile occurrences, we choose \( C_k \) so that only a small percentage (e.g. 1–3%) of tiles have high-quality multiplicity greater than \( C_k \). Next, if \( C_k \) is chosen so that a larger percentage (e.g. 4–6%) of tiles occur more than \( C_k \) times in \( R \) As \( C_k \) value decreases, more errors are corrected at the cost of an increased risk of false error correction. The specific values chosen depend on the histogram of tile occurrences. By default, we set \( C_k = 2 \) such that a low frequency tile could only be corrected to a tile with at least twice the frequency. Increasing \( C_k \) improves the confidence in error correction. Finally, we choose \( k = \log (|t|) \) when an estimate of the length of the genome is available, otherwise, a number between 10 and 16 should work. Tile size is \( \sim 2k \), so k-mer overlap is 0 to a few bases. The maximum Hamming distance \( d \) is set to one by default. But when \( k \) is chosen to be relatively large (e.g. 14 to 16), increasing \( d \) allows us to identify more sequencing errors but incurs a longer run time and increases the risk of false error prediction.

3.1.4 Overall complexity. Combining the analysis for each step, the overall run time is \( O(nR^2 \log |R|) \) and the space usage is \( O(8|R| + |R|^2) \).

When the collection of input short reads \( R \) does not fit in main memory, we propose a divide and merge strategy where \( R \) is partitioned into chunks small enough to occupy just a portion of main memory. For each chunk, we stream through each read and record the k-spectrum and tile information, merging it with the data from previous chunks. Reads need not be stored in memory after they have been processed. A similar strategy is applied for error correction: \( R \) is reloaded into memory in chunks, tilings and d-mutant tilings are inferred for each read, and errors are corrected as warranted.

4 RESULTS

We evaluated Reptile on several Illumina/Solexa datasets and compared the results with SHREC (Schröder et al., 2009) version 2.0, a recent high-quality short-read error correction method that is itself shown to give superior results over prior k-spectrum approaches. We omitted evaluation on simulated data because the model assumptions used in the simulations with random errors or synthetic genomes do not accurately reflect actual short-read sequencing errors (Dohm et al., 2008), and could even be misleading. Our test datasets are Illumina-generated short reads of well-characterized, Sanger assembled bacterial genomes. Knowledge of the genomes is needed for determining the accuracy of the error correction methods. The six experimental datasets, downloaded from the sequence read archive at NCBI, are listed in Table 1. Datasets D1 (Accession Number: SRX000429), D2 (SRR016653_1), D5 (SRR022918_1) and D6 (SRR034599_1) are Illumina reads from the E.coli strain K-12 substr (NC_000913) genome (\( \sim 4.64 \) Mbp); datasets D3 (SRR006332) and D4 are Illumina reads from the Acinetobacter sp. ADP1 (NC_005966) genome (\( \sim 3.6 \) Mbp). The first four datasets are generated by Solexa 1G Genome Analyzer, where each read has the same length 36 bp. The latter two datasets are generated using the more recent Illumina Genome Analyzer II, with read lengths of 51 bp in D5 and 101 bp in D6. D1 has high coverage and low error rate. D2 has typical coverage and low error rate. D3 has high coverage and high error rate. D4 is derived from D3 by randomly selecting short reads amounting to \( 40 \times \) coverage. This is done for evaluating performance on a low coverage, high error rate dataset. Both D5 and D6 have higher error rates. In addition, \( >13.9 \% \) of the reads in D6 contain ambiguous nucleotides, denoted by character N. Since SHREC cannot process non-ACGT characters, we eliminated all reads with ambiguous bases, even though Reptile has no such limitation. The number of discarded reads is indicated in column 5, Table 1.

Similar to Schröder et al. (2009), we evaluated error correction results with the aid of RMAP (v2.05) (Smith et al., 2008), which maps short reads to a known genome by minimizing mismatches. We allowed up to five mismatches per read in the first four datasets.
Table 2. Experimental datasets

<table>
<thead>
<tr>
<th>Dataset</th>
<th>Genome</th>
<th>Read length (bp)</th>
<th>Number of reads (M)</th>
<th>Discarded reads</th>
<th>Cov. Error rate (%)</th>
</tr>
</thead>
<tbody>
<tr>
<td>D1</td>
<td>E. coli</td>
<td>36</td>
<td>20.8</td>
<td>107.7 K</td>
<td>160x</td>
</tr>
<tr>
<td>D2</td>
<td>E. coli</td>
<td>36</td>
<td>10.4</td>
<td>48.3 K</td>
<td>80x</td>
</tr>
<tr>
<td>D3</td>
<td>A. sp.</td>
<td>36</td>
<td>17.7</td>
<td>456 K</td>
<td>173x</td>
</tr>
<tr>
<td>D4</td>
<td>A. sp.</td>
<td>36</td>
<td>4.0</td>
<td>0</td>
<td>40x</td>
</tr>
<tr>
<td>D5</td>
<td>E. coli</td>
<td>47</td>
<td>7.0</td>
<td>32.7 K</td>
<td>71x</td>
</tr>
<tr>
<td>D6</td>
<td>E. coli</td>
<td>101</td>
<td>8.9</td>
<td>1.44 M</td>
<td>193x</td>
</tr>
</tbody>
</table>

Error rate is estimated by mapping the reads to the corresponding genome using RMAP, and finding mismatches based on uniquely mapped reads.

Table 2. Results of mapping each dataset to the corresponding genome using RMAP

<table>
<thead>
<tr>
<th>Dataset</th>
<th>Allowed mismatches</th>
<th>Number of reads a</th>
<th>Uniquely mapped reads (%)</th>
<th>Ambiguously mapped reads (%)</th>
</tr>
</thead>
<tbody>
<tr>
<td>D1</td>
<td>5</td>
<td>20708709</td>
<td>96.5</td>
<td>2.5</td>
</tr>
<tr>
<td>D2</td>
<td>5</td>
<td>10359952</td>
<td>96.7</td>
<td>2.5</td>
</tr>
<tr>
<td>D3</td>
<td>5</td>
<td>17675271</td>
<td>79.9</td>
<td>1.5</td>
</tr>
<tr>
<td>D4</td>
<td>5</td>
<td>40000000</td>
<td>84.1</td>
<td>1.6</td>
</tr>
<tr>
<td>D5</td>
<td>10</td>
<td>7049153</td>
<td>62.5</td>
<td>1.5</td>
</tr>
<tr>
<td>D6</td>
<td>10</td>
<td>8874761</td>
<td>63.5</td>
<td>1.2</td>
</tr>
</tbody>
</table>

aNumber of reads containing no ambiguous bases.

Table 2 shows results of mapping each dataset to the corresponding genome using RMAP, and finding mismatches based on uniquely mapped reads.

and allowed up to 10 mismatches (default value of RMAP) in D5 and fifteen mismatches in D6 since the reads are longer in the latter two datasets. Reads that could not be mapped to the genome, or that map to multiple locations, are discarded. The mismatches between uniquely mapped reads and the genome are considered read errors. Quality of the datasets varied as shown in Table 2, with the percentage of reads that are uniquely mapped ranging from 62.5 to 96.7%. The large percentage of unmappable reads, the higher error rates as well as the large percentage of reads with ambiguous bases indicate that D5 and D6 have lower quality than D1 to D4.

Since the goal of error correction is to identify and correct each erroneous nucleotide, we assess the quality of error correction at the base level. A true positive (TP) is any erroneous base that is changed to the true base, a false positive (FP) is any true base changed wrongly, a true negative (TN) is any true base left unchanged, and a false negative (FN) is any erroneous base left unchanged. Then Sensitivity = TP/(TP + FN) and Specificity = TN/(TN + FP). Note that these definitions are different from those used by Schröder et al. (2009), which target read-level error detection (whether a read is flagged as containing an error or not). This is a less stringent measure because any read containing errors was classified as TP provided at least one of its errors was detected and irrespective of whether they were accurately corrected or not.

We propose two additional measures for assessing the quality of error correction:

- **Error rate**: The number of errors in the dataset divided by the total number of bases in the dataset.
- **Gain**: Gain is calculated as (TP – FN)/(TP + FN). This measures the percentage of errors effectively removed from the dataset, which is equivalent to the number of errors before correction minus the number of errors after correction divided by the number of errors before correction. Clearly, Gain should approach one for the best methods, but may be negative for methods that actually introduce more errors than they correct.

We regard these measures as important because they penalize failing to detect an erroneous base, correctly detecting an erroneous base but wrongly correcting it, and characterizing a correct base to be an erroneous base. In particular, we strongly advocate the Gain measure as it captures data quality post-error correction compared with the quality prior to the correction.

The results of running Reptile and SHREC on the six datasets are summarized in Table 3. Due to the larger memory usage of the SHREC program, we were not able to obtain results for D3, D5 and D6. In all other cases, Reptile had higher Gain and lower EBA than SHREC. With other parameters fixed in Reptile, we varied the parameter $d$ to $d'$. As expected, the run time significantly increased as $d'$ increased, since the size of $d$-neighborhood for each kmer increased. Also, we see an increase in both TP and FP and four to five times higher EBA, indicating that when we increase the search space, we run the risk of false error detection and correction but increase the chance to identifying more errors.

An inherent difficulty in using any method is the challenge of choosing optimal parameters. The results reported in Table 3 are obtained when using the parameter choices suggested in Section 3. To show that even better performance is possible, we applied a series of parameter choices to dataset D3 (Fig. 3). Gain improved from 63% with the default parameters to as high as 72%. We chose to report on Reptile using the default parameters for all cases in Table 3 because it is unfair to choose optimal parameters for each individual case based on our knowledge of the genome, which would generally not be known. Similarly, we used the default parameter settings for SHREC. Using a different combination of parameters may vary the results of both SHREC and Reptile. In this article, we have presented a method to select parameters for Reptile based on known quantities such as kmer frequency and quality score histograms. A similar guidance is needed for the SHREC program and is beyond the scope of the article. Note that we do not take into account improved results that can only be obtained by the knowledge of the genome (Fig. 3). One can observe that our method of parameter estimation based on statistics from the dataset is performing better than analytical calculations based on the assumptions of uniform error distribution and uniform coverage of genome by reads.

In addition, we compared the run time and memory usage of SHREC and Reptile. SHREC is a multi-threaded program while the current release of Reptile can only use a single core. Hence, we report run times in total CPU hours. Both methods were run on a SUN Fire X2200 workstation with dual quad-core 2.3 GHz AMD Barcelona three processors with 8 GB RAM and 4 GB swap memory, running Debian GNU/Linux x86_64. Results in Table 3 show Reptile is 3–10 times faster and uses 8–11 times less memory than SHREC. As expected, memory usage of Reptile is associated with the length of the genome and the number of errors in the data.
Table 3. Results of Reptile and SHREC on Illumina sequenced short reads

<table>
<thead>
<tr>
<th>Data</th>
<th>Method</th>
<th>TP</th>
<th>FN</th>
<th>FP</th>
<th>TN</th>
<th>EBA (%)</th>
<th>Sensitivity (%)</th>
<th>Specificity (%)</th>
<th>Gain (%)</th>
<th>CPU Hours</th>
<th>Memory (GB)</th>
</tr>
</thead>
<tbody>
<tr>
<td>D1</td>
<td>SHREC</td>
<td>2519754</td>
<td>1183861</td>
<td>667435</td>
<td>740842474</td>
<td>1.794</td>
<td>70.4</td>
<td>99.9</td>
<td>53.8</td>
<td>-</td>
<td>&gt;8</td>
</tr>
<tr>
<td>(160x)</td>
<td>Reptile (1)</td>
<td>3164394</td>
<td>839221</td>
<td>133558</td>
<td>741376351</td>
<td>0.007</td>
<td>79</td>
<td>99.9</td>
<td>75.7</td>
<td>0.79</td>
<td>1.1</td>
</tr>
<tr>
<td></td>
<td>Reptile (2)</td>
<td>3457177</td>
<td>545898</td>
<td>245417</td>
<td>741264492</td>
<td>0.028</td>
<td>86.4</td>
<td>99.9</td>
<td>80.2</td>
<td>2.49</td>
<td>1.1</td>
</tr>
<tr>
<td>D2</td>
<td>SHREC</td>
<td>1305505</td>
<td>422337</td>
<td>251228</td>
<td>370981202</td>
<td>1.549</td>
<td>75.5</td>
<td>99.9</td>
<td>61.0</td>
<td>3.6</td>
<td>7.1</td>
</tr>
<tr>
<td>(79.5x)</td>
<td>Reptile (1)</td>
<td>1169256</td>
<td>556586</td>
<td>44959</td>
<td>371187471</td>
<td>0.009</td>
<td>67.8</td>
<td>99.9</td>
<td>65.2</td>
<td>0.35</td>
<td>0.84</td>
</tr>
<tr>
<td></td>
<td>Reptile (2)</td>
<td>1315277</td>
<td>410565</td>
<td>91205</td>
<td>371141225</td>
<td>0.042</td>
<td>76.2</td>
<td>99.9</td>
<td>70.9</td>
<td>1.23</td>
<td>0.84</td>
</tr>
<tr>
<td>D3</td>
<td>SHREC</td>
<td>7138883</td>
<td>2361813</td>
<td>1138666</td>
<td>610144638</td>
<td>0.013</td>
<td>75.1</td>
<td>99.8</td>
<td>63.2</td>
<td>1.66</td>
<td>2.2</td>
</tr>
<tr>
<td>(172.5x)</td>
<td>Reptile (1)</td>
<td>1473252</td>
<td>530736</td>
<td>61921</td>
<td>141382091</td>
<td>1.306</td>
<td>73.5</td>
<td>99.6</td>
<td>42.9</td>
<td>2.78</td>
<td>7.6</td>
</tr>
<tr>
<td>D4</td>
<td>SHREC</td>
<td>1422409</td>
<td>581039</td>
<td>222218</td>
<td>141773794</td>
<td>0.091</td>
<td>71</td>
<td>99.8</td>
<td>59.9</td>
<td>0.26</td>
<td>0.66</td>
</tr>
<tr>
<td>(40x)</td>
<td>Reptile (1)</td>
<td>3457717</td>
<td>545898</td>
<td>245417</td>
<td>741264492</td>
<td>0.028</td>
<td>86.4</td>
<td>99.9</td>
<td>80.2</td>
<td>2.49</td>
<td>1.1</td>
</tr>
<tr>
<td></td>
<td>Reptile (2)</td>
<td>3551764</td>
<td>3189748</td>
<td>985674</td>
<td>323583005</td>
<td>0.017</td>
<td>52.7</td>
<td>99.7</td>
<td>38.1</td>
<td>0.94</td>
<td>1.9</td>
</tr>
<tr>
<td>D6</td>
<td>SHREC</td>
<td>1315277</td>
<td>410565</td>
<td>91205</td>
<td>371141225</td>
<td>0.042</td>
<td>76.2</td>
<td>99.9</td>
<td>70.9</td>
<td>1.23</td>
<td>0.84</td>
</tr>
<tr>
<td>(193x)</td>
<td>Reptile (1)</td>
<td>17158925</td>
<td>2947342</td>
<td>1298891</td>
<td>874945703</td>
<td>0.010</td>
<td>85.3</td>
<td>99.9</td>
<td>78.9</td>
<td>2.76</td>
<td>4.6</td>
</tr>
</tbody>
</table>

Fig. 3. Gain and sensitivity versus different parameter choices for D3. The first 11 sample points use parameters \(k = 11, d = 1, |t| = 22\), and \((C_m, Q_e)\) values (14, 60), (12, 60), (10, 60), (10, 55), (8, 60), (8, 55), (7, 45), (6, 45), (5, 45), respectively. The last sample point uses parameters \(k = 12, d = 2, |t| = 24, C_m = 8, Q_e = 45\).

The above experiments were carried out by excluding all reads containing ambiguous bases. However, Reptile does have functionality to deal with ambiguous bases, which is useful in the following cases: (i) if a read contains few ambiguous bases, the surrounding high-quality regions provide sufficient information to infer correct bases by referencing the \(k\)-spectrum; (ii) in some datasets, neglecting reads containing ambiguous bases leads to excessive loss of data, which further distorts uniformity of the sampling. For instance, as much as 13.9% of reads in dataset D6 contain \(N\)’s.

Reptile attempts to correct ambiguous bases in regions where their density is low. If a read contains too many ambiguous bases, it is low quality and untrustworthy. Some reads may have ambiguous bases clustered in some region, e.g. the 3’ region, while other parts may still be of good quality. It is more meaningful to try correcting ambiguous bases in the latter parts alone, since a cluster of ambiguous nucleotides in a read makes it unlikely to pinpoint other reads that have the same genomic co-location. Formally, Reptile attempts to correct an ambiguous base \(b\) of read \(r\), if in any substring \(r[i:|i|+w]\) that contains \(b\), there are no more than \(d\) ambiguous bases. The ratio of \(d\) to \(w\) constrains the maximum density of ambiguous bases allowed in attempting error correction. By default, \(d\) is set to \(k\) (to equal \(k\)-mer length), while \(d\) is set to the maximum Hamming distance allowed (Section 3). To implement the above idea, all ambiguous bases satisfying the density constraint are changed to one of the bases from the set \{A, C, G, T\} initially (default ‘A’), and will be validated or corrected later by the algorithm.

To test the accuracy of this procedure as well as study the effect of the choice of the default base, we conducted Reptile runs on the full datasets of \(D_2\) (36 bp) and \(D_6\) (101 bp) by setting the ambiguous bases to the chosen default. The results are presented in Table 4. The default base used is shown under Column ‘N’. Accuracy is defined to be the percentage of ambiguous bases that have been successfully corrected (again, only reads uniquely mapped by RMAP are considered as truth is unknown otherwise). The last four
The proposed error correction algorithm is conservative because it avoids changing bases unless there is a compelling under-estimation of errors in read representation of a tile compared with its correct version. This is likely to avoid under-estimation of the sensitivity, although we considered 5 here and tested as many as 20 substitutions, typically just two, with full chance that the true number of errors is less than the minimum.

There are natural polymorphisms among bacterial species and there remain several additional challenges in next-generation sequencing error correction. One challenge is to distinguish errors from polymorphisms, for example, single nucleotide polymorphisms (SNPs). Reptile could accommodate SNP prediction with modification in the tile correction stage (Algorithm 1), where ambiguities may indicate polymorphisms. Another challenge is the growing read length of upcoming high-throughput sequencers. Currently, we define tiles as concatenations of two k-mers. It might prove useful to extend the tile definition to more than two k-mers in order to address error correction in much longer reads.

Conflict of Interest: none declared.

REFERENCES