Sequence analysis

Reptile: representative tiling for short read error correction

Xiao Yang1,2, Karin S. Dorman2,3,4 and Srinivas Aluru1,2,4,5,*

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

ABSTRACT

Motivation: Error correction is critical to the success of next-generation sequencing applications, such as resequencing and 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.

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 k-mers from the input reads, and corrects errors by simultaneously examining: (i) Hamming distance-based correction possibilities for potentially erroneous k-mers; and (ii) neighboring 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.

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

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, de novo genome assembly, metagenomics and beyond (Anzorge, 2009; Marguerat et al., 2008). 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 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 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 MiSeq (Tamm 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 k-mer is considered solid if its multiplicity exceeds a threshold, and insolid otherwise. Reads containing insolid k-mers are corrected using a minimum number of edit operations so that they contain only solid 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 transcriptome 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 up to 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 rarely produce by short-read sequencing technology (Dohm et al., 2008) of \( r \) with substitution errors, assuming insertion and deletion errors are negligible. 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, 2009) and even in high coverage situations, the occurrence of many exactly coincident reads, e.g., \( r_0 \) and \( r_1 \) in Figure 1, are rare. We therefore resort to alignments on subreads, the substrings of a read.

Storing \( R \) in its entirety is memory intensive, so it is less compute intensive 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 subread \( a_2' \) reduces alignment time. To identify k-mers instances in larger datasets. As with most current approaches including FreClu, Reptile is targeted to short reads with substitution errors, assuming insertion and deletion errors are rarely produced by short-read sequencing technology (Dohm et al., 2008).

2 NOTATIONS

Let \( R = \{r_1, r_2, \ldots, r_n\} \) 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 \( \text{Cov} = \sum_i r_i^L \), where \( \text{Cov} \) denotes the genome length.

Define the \( k \)-spectrum of a read \( r \) to be the set \( R_k = \{r[i : i + k - 1] | 0 \leq i < L - k + 1\} \), 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 \( R_k = \bigcup_{r \in R} R_k \).

Let \( \alpha \) and \( \beta \) be two strings such that \( \alpha[0] = \beta[0] \) and \( \alpha[-1] = \beta[-1] \). We define the \( l \)-concatenation of \( \alpha \) and \( \beta \), denoted \( \alpha \circ \beta \), to be the string \( \gamma \) of length \( l + |\alpha| + |\beta| - l \) such that \( \gamma[0 : l] = \alpha[l] \) and \( \gamma[l + |\beta| - l] = \beta \).

The Hamming distance between two strings \( \alpha_1 \) and \( \alpha_2 \) for \( |\alpha_1| = |\alpha_2| \), denoted \( d_h(\alpha_1, \alpha_2) \), is the number of positions at which they differ.

3 METHODS

The success of any error correction method relies on an adequate coverage of the target genome. If we knew 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 \( T \) 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, 2009) and even in high coverage situations, the occurrence of many exactly coincident reads, e.g., \( r_0 \) and \( r_1 \) in Figure 1, are rare. We therefore resort to alignments on subreads, the substrings of a read.

Storing \( R \) in its entirety is memory intensive, so it is less compute intensive 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 subread \( a_2' \) reduces alignment time. To identify k-mers instances in larger datasets. As with most current approaches including FreClu, Reptile is targeted to short reads with substitution errors, assuming insertion and deletion errors are rarely produced by short-read sequencing technology (Dohm et al., 2008).

3 METHODS

The success of any error correction method relies on an adequate coverage of the target genome. If we knew 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 \( T \) 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, 2009) and even in high coverage situations, the occurrence of many exactly coincident reads, e.g., \( r_0 \) and \( r_1 \) in Figure 1, are rare. We therefore resort to alignments on subreads, the substrings of a read.

Storing \( R \) in its entirety is memory intensive, so it is less compute intensive 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 subread \( a_2' \) reduces alignment time. To identify k-mers instances in larger datasets. As with most current approaches including FreClu, Reptile is targeted to short reads with substitution errors, assuming insertion and deletion errors are rarely produced by short-read sequencing technology (Dohm et al., 2008).

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, 2009) and even in high coverage situations, the occurrence of many exactly coincident reads, e.g., \( r_0 \) and \( r_1 \) in Figure 1, are rare. We therefore resort to alignments on subreads, the substrings of a read.

Storing \( R \) in its entirety is memory intensive, so it is less compute intensive 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 subread \( a_2' \) reduces alignment time. To identify k-mers instances in larger datasets. As with most current approaches including FreClu, Reptile is targeted to short reads with substitution errors, assuming insertion and deletion errors are rarely produced by short-read sequencing technology (Dohm et al., 2008).

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, 2009) and even in high coverage situations, the occurrence of many exactly coincident reads, e.g., \( r_0 \) and \( r_1 \) in Figure 1, are rare. We therefore resort to alignments on subreads, the substrings of a read.

Storing \( R \) in its entirety is memory intensive, so it is less compute intensive 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 subread \( a_2' \) reduces alignment time. To identify k-mers instances in larger datasets. As with most current approaches including FreClu, Reptile is targeted to short reads with substitution errors, assuming insertion and deletion errors are rarely produced by short-read sequencing technology (Dohm et al., 2008).
Nevertheless, relying solely on short k-mers can easily lead to ambiguities when resolving erroneous bases. For instance, in Figure 1, without knowing the alignment, it is unclear if \(a_1\) should be corrected to \(x\) or \(y\), since both \(h(x|a_2) = h(y|a_2)\) and \(a_2\) and \(a_1\) have a similar higher frequency. However, the contextual information of \(a_2\) available from read \(r_1\) (in this case, \(a_1\)) uniquely identifies \(a_2\) 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 contigs co-exist in observed reads. For instance, \(u_1||u_2\) exists in \(r_1\), \(r_2\), or \(r_3\), and \(u_1||u_2\) exists in \(r_3\) in Figure 1. The following definitions formalize the notion of contextual information.

**Definition 1.** \(r = u_1||u_2\) (0 ≤ \(i < k\)) is a tile of read \(r\) if \(t\) is a substring of \(r\), and \(|t| = |u_1| = |u_2| = k\).

**Definition 2.** \(r' = u'\|f\) is a d-mutant tile of \(r = u_1||u_2\) if \(h(u'|f)\leq d\) and \(h(f|f')\leq d\).

**Definition 3.** \(T_{r}(t_1,t_2,...,t_n)\) is a tiling of read \(r\) if \(r = t_1||t_2||...||t_n\), such that \((1)\ t_1\ (1 \leq i \leq n)\ is a tile of \(r\) and (2) \(r_1\leq i < n\).

Note that if \(t_i\) is specified as \(t(x|y)_j\), then the overlaps between consecutive tiles can be inferred; i.e. \(t(x'|y)_{j+1}\) can be derived from \(t(x|y)_j\). Multiple tiles exist for any read. For example, both \(t(x|y)_j(t(x'|y)_{j+1})\) and \(t(x|y)_j(t(x'|y)_{j+1})\) are tiles of \(r\). We also extend the concept of d-mutant tiles to tiles. For instance, we can think of \((t(x|y)_j(t(x'|y)_{j+1}))\) as a one-mutant tiling of \(T_{r}(t_1,t_2,...,t_n)\). Similarly, \((t(x|y)_j(t(x'|y)_{j+1}))\) is a two-mutant tiling of \(T_{r}(t_1,t_2,...,t_n)\). Formally,

**Definition 4.** A d-mutant tiling of \(T_{r}(t_1,t_2,...,t_n)\) is a sequence of tiles \((t_1,t_2,...,t_n)\) such that \((1)\ |t_i| = |t| \leq d\) and \(h(t|t')\leq d\), where \(t_i\) is implicitly determined by \(r\) for \(1 \leq i \leq m\), and \((2)\ t_j\ is a d-mutant tile of \(t_i\) for \(1 \leq j \leq m\).

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

### 3.1 The algorithm

#### 3.1.1 Phase 1: information extraction

**Constructing \(R^d\) involves one linear scan of each read in \(R\). This takes \(O(n)\) time.** We maintain \(R^d\) in sorted order using \(O(|R^d|\log|R^d|)\) time. The space requirement for \(R^d\) is given by \(|R^d| = \text{Min}(d^2, \text{HL} - k + 1)\). 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 k-mer, ideally in constant time with respect to \(k\). One could do so by storing the entire Hamming graph \(G_{2^d}\), but it would require large amount of memory. If we assume \(G\) as a random string, and errors accumulate independently with probability \(p_k\), then the probability that a node is linked to another is \(p_k = \sum_{j=0}^{\min(k,d)}\binom{d}{j}\frac{p_j}{2^d}\cdot\frac{p^{d-j}}{2^d}\), including the chance that another random k-mer in the genome is within d-Hamming distance of the current k-mer and the chance that the current k-mer contains up to d errors. Thus, the expected memory usage is \(O\left(\left(\binom{d}{2}\right)\cdot|p_k|\right)\).

Alternatively, we could keep all the d-mutant tiles associated with a given k-mer \(u_i\) by checking whether each k-mer in its complete neighborhood, \(u_i\|u_0\), exists in \(R^d\). If \(u_i\) \(\in\) \(R^d\), then there is an edge between \(v_i\) and \(v_j\) in the Hamming graph. This takes \(O(|R^d|\log|R^d|)\) time using binary search. There are \((\sum_{j=0}^{d}\binom{d}{j})\) possible candidate k-mers for each \(u_i\) \(\in\) \(R^d\), so it takes \(O\left(\left(\binom{d}{2}\right)\cdot|p_k|\right)\) time to identify all edges.

To reduce the average time for inferring \(N^d\) of \(u_i\), we replicate \(R^d\) in memory and sort each replicate on a different subset of positions in the k-mer string, using the following strategy:

- (a) Store indices 0 to \(k - 1\) in a vector \(A\).
- (b) Divide \(A\) evenly into \(c\) \((d < c \leq k)\) chunks, each of size \(|k/c|\) or \(|k/c|+1\).
- (c) For every choice of \((\text{chunk})\) chunks, sort \(R^d\) by masking the indices from selected chunks and store the sorted results separately.

To identify \(N^d\) of \(u_i\), we query \(u_i\) against each sorted k-spectrum \(R^d\ (0 \leq j < \binom{d}{2})\) by binary search considering only indices used for sorting \(R^d\). All k-mers that belong to the d-neighborhood of \(u_i\) are adjacent to \(u_i\) in at least one \(R^d\). If sequencing errors accumulate independently, then the expected number of elements of \(N^d\) found in every \(R^d\) is \(e^{-\frac{d^2}{4}\cdot\text{HL}}\). Hence, we need approximately \(\left(\frac{d^2}{4}\cdot\text{HL}\right)\) expected time to recover all edges of the Hamming graph, i.e. \(O\left(|R^d|\log|R^d|\right)\) time assuming both \(\left(\binom{d}{2}\right)\) and \(c\) are constants. Typically, \(\left(\binom{d}{2}\right) < 4^d\), therefore, choosing a larger \(\left(\binom{d}{2}\right)\) 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 \(R^d\) required only ~560 MB memory, but the average number of hits per 13-mer in each 13-spectrum was less than one. Therefore, identifying each element of \(N^d\) 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 \(N^d\) by inferring \(N^d\) for every element in \(N^{d-1}\). This strategy might be more biologically meaningful (Qu et al., 2009), but is only an approximation since an edge between two vertices \(v_i\) and \(v_j\) could be recovered only if there exists a path connecting them such that adjacent vertices represent k-mers 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 k-mers found in reads. Here, we use one fixed value of \(l\) but several 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 the collected tiles and one linear scan of the sorted list. This process takes \(O(|R^d|\log|R^d|)\) time, where \(|R^d| = \text{Min}(2^d,\text{HL} - 2k + 1))\). Meanwhile, we record the number of occurrences of each tile, where every position has a quality score exceeding some threshold \(Qc\). Typically, a quality score is associated with every base of a short read. The score indicates the probability \(p_k\) that the corresponding base is sequenced incorrectly.
For instance, Illumina Genome Analyzer 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 of the tiling graph takes twice the time, but no additional memory is needed since \( R^2 \) is already generated using both strands.

3.1.2 Phase 2: error correction We use the contextual information in read \( R \) to identify sequencing errors through the process of choosing a tiling \( T_r \), and comparing it with its \( d \)-mutant tilings. In particular, if \( R \) contains \( s \) errors, and we choose any tiling \( T_r \), then an error-free tiling \( T_r \) belongs to the collection of \( d \)-mutant tilings of \( T_r \), if \( d \geq s \). Under the standard assumption of uniform coverage, the tiling of \( T_r \) should be substantially more abundant than at least some of the tilings of \( T_r \) with errors. After \( T_r \) is identified, the true read \( r \) can be readily inferred from \( T_r \).

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 \( T_r \) is not one of the \( d \)-mutant tilings of \( T_r \). On the other hand, an alternate tiling \( V_r \) of \( r \) may lower the maximum number of mutations per k-mer to below \( d \) such that \( V_r \) with high frequency tilings is one of the \( d \)-mutant tilings of \( V_r \). In the case that there is no such tiling \( V_r \), we examine a subset of constituent tiles in \( V_r \). 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 \( t \) on \( r \) and attempt to correct \( r \) via comparisons with its \( d \)-mutant tiles (tile correction). If \( t \) is validated or corrected, move to the next tile in the standard tiling and repeat. If \( t \) 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 \( d \) neighborhoods. 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 comparison of 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 its occurrence count exceeds an upper threshold \( C_t \) (lines 1–2). A low 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 tiles of \( t \) with much higher frequencies (at least \( C_t \) times, \( C_t > 1 \) than \( t \) and which differ at low-quality bases \(< Q_o \) in \( t \), it is likely that \( t \) contains error(s) and the true 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. Similar reasoning applies to \( t \) 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 \( t \) and \( t' \in T \), 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 \( r \), 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 tile of \( t_1 \) lead to an inconclusive decision. Hence, \( t_1 \) is not selected in the tiling and an 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 \( d_2 \). 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 \( 5' \). 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 \( L/\theta \). Hence, the time spent in correcting each read is \( O(\log(nL)) \), and the overall run time of Algorithm 2 is \( O(dL\log(nL)) \).

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 \( Q_o \) such that a given percentage (e.g. 15–20%) of bases have quality score below \( Q_o \). This value could

Algorithm 1 Tile \( t \) error correction.

1. if \( Q(t) \geq C_t \) then
   1.1. \( t \) is valid; return.
   end if
2. if \( t \) has no \( d \)-mutant tiles \( t' \) then
   2.1. if \( Q(t') \geq C_m \) then
   2.1.1. \( t \) is valid; return.
   end if
   2.2. \( t \) is not unique, return due to insufficient evidence.
   end if
3. if \( Q(t) \geq C_m \) then
   3.1. \( t \) is validated or corrected using Algorithm 1 (line 4). The new placement of \( t_{ \text{new} } \) 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 tile of \( t_1 \) lead to an inconclusive decision. Hence, \( t_1 \) is not selected in the tiling and an 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 \( d_2 \). 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 \( 5' \). 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 \( L/\theta \). Hence, the time spent in correcting each read is \( O(\log(nL)) \), and the overall run time of Algorithm 2 is \( O(dL\log(nL)) \).
Algorithm 2 Read error correction.

1. Initialize: \( t \leftarrow t_0 \), where \( t_0 \) is a prefix of \( r \); \( d_1 \leftarrow d \).
2. while \( t \neq \emptyset \) do
3. Identify \( d \)-mutant tiles of \( t = a_1 \| a_2 \| a_3 \) as the set \( \{t' = a_1' \| a_2' \| a_3' \in N_1^* \times N_2^* \times N_2^* \} \).
4. Correct \( t \) (Algorithm 1).
5. Select the decisions made on \( t \) in the former step, tile \( t_{\text{sel}} \) of \( r \) will be chosen according to \([D1]–[D3]\) as follows. If there is insufficient space to place a tile toward the end of \( r \), \( t_{\text{sel}} \) will be chosen as the suffix of \( r \).
6. \([D1]\) is valid: select \( t_{\text{sel}} \) such that the suffix–prefix overlap between \( r \) and \( t_{\text{sel}} \) equals \( a_2' \), \( d_1 \leftarrow 0 \).
7. \([D2]\) was corrected to \( t' = a_1' \| a_2' \| a_3' \) and \( t \) is replaced with \( t' \) in \( r \); select \( t_{\text{sel}} \) such that the suffix–prefix overlap between \( r \) and \( t_{\text{sel}} \) equals \( a_2' \), \( d_1 \leftarrow 0 \).
8. \([D3]\) Insufficient evidence to correct: select \( t_{\text{sel}} \) to be one of the following. Let \( r[1..i] \) be a maximal validated or corrected region of \( r \) that overlaps with the \( 5' \) region of \( t \), where \( i_2 \leq i \leq |r| \) and \( i_1 + 1 \leq i \); \( r[1..i] \) is a maximal uncorrected region from previous iterations due to insufficient evidence.
   a. if \( r[i_1 + 1..i] = \emptyset \), then \( t_{\text{sel}} \leftarrow t[-i_2+2..i_2+1] \) and \( d_1 \leftarrow 1 \).
   b. if \( r[i_1 + 1..i] \neq \emptyset \), then \( t_{\text{sel}} \leftarrow r[i_1 + 1..i] \).

Similarly, if \( r[i_1..i_2] \) overlaps with \( 3' \) end of \( r \) and \( r[i_1..i_1-1] \) is a maximal uncorrected region from previous iterations, then if \( r[i_1..i_1-1] = \emptyset \), set \( t_{\text{sel}} \leftarrow r[i_1..i_1-1] \) and \( d_1 \leftarrow 2 \).

In above cases, \( t_{\text{sel}} \) is valid only if it was not assigned with the same value in previous iterations (unless \( t_{\text{sel}} \) became the suffix of \( r \)). If such a choice does not exist, \( t_{\text{sel}} = \emptyset \).
9. \( t \leftarrow t_{\text{sel}} \).
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-concatenation of two adjacent kmers. 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 \( Q_k \) 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 \( C_k \). \( C_{\text{m}} \) is chosen so that a larger percentage (e.g. 4–6%) of tiles occur more than \( C_{\text{m}} \) times in \( R \). As \( C_{\text{m}} \) 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 \( C_{\text{m}} = 2 \) so that a low frequency tile could only be corrected to a tile with at least twice the frequency. Increasing \( C_{\text{m}} \) improves the confidence in error correction. Finally, we choose \( k = \lceil \log_2(nR) \rceil \) 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 \( O(nR^2) + |R|^3 \) and the space usage is \( O(|R| + |R|^2) \) 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, tiling 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 k-k-spectrum limitations 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 (SRR001665_1), D3 (SRR022918_1) and D6 (SRR034509_1) are Illumina reads from the E.coli str. K-12 substr (NC_000913) genome (∼4.64 Mbp), datasets D3 (SRR006332) and D4 are Illumina reads from the Acinetobacter sp. ADP1 (NC_005966) genome (∼3.6 Mb). 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 47 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 40x 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 (v0.5) (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.
Experimental datasets

Table 1. Experimental datasets

<table>
<thead>
<tr>
<th>Data</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>160× 0.6</td>
</tr>
<tr>
<td>D2</td>
<td>A. sp.</td>
<td>36</td>
<td>10.4</td>
<td>48.3 K</td>
<td>80× 0.6</td>
</tr>
<tr>
<td>D3</td>
<td>A. sp.</td>
<td>36</td>
<td>17.7</td>
<td>456 K</td>
<td>173× 1.5</td>
</tr>
<tr>
<td>D4</td>
<td>A. sp.</td>
<td>36</td>
<td>4.0</td>
<td>0</td>
<td>40× 1.5</td>
</tr>
<tr>
<td>D5</td>
<td>E. coli</td>
<td>47</td>
<td>7.0</td>
<td>32.7 K</td>
<td>71× 3.3</td>
</tr>
<tr>
<td>D6</td>
<td>E. coli</td>
<td>101</td>
<td>8.9</td>
<td>1.44 M</td>
<td>193× 2.2</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>Data</th>
<th>Allowed mismatches</th>
<th>Number of reads</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>5</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>
<tr>
<td></td>
<td>15</td>
<td></td>
<td>68.8</td>
<td>1.4</td>
</tr>
</tbody>
</table>

*Number of reads containing no ambiguous bases.

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:

- **Errorneous base assignment (EBA):** let \( n_e \) denote the number of erroneous bases that are correctly identified but changed to a wrong base. Then, \( \text{EBA} = n_e/(TP + n_e) \) reflects how well we are able to correct an erroneous base to the true base after a sequencing error has been identified. A lower value of EBA indicates a more accurate base assignment.
- **Gain:** \((TP – FP)/(TP + FN)\). 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 maximum \( d \) value used for inferring Hamming graph in D1 and D2. As expected, the run time significantly increased as \( d \) increased, since the size of \( d \)-neighborhood for each k-mer 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 identify 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 k-mer 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 (Cov)</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 (160x)</td>
<td>SHREC</td>
<td>2819754</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></td>
<td>Reptile (1)</td>
<td>3164934</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>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>D2 (79.5x)</td>
<td>SHREC</td>
<td>1305005</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></td>
<td>Reptile (1)</td>
<td>1160256</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 (172.5x)</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></td>
<td>Reptile (1)</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>D4 (40x)</td>
<td>SHREC</td>
<td>1473252</td>
<td>530736</td>
<td>251228</td>
<td>141382091</td>
<td>1.506</td>
<td>73.5</td>
<td>99.6</td>
<td>42.9</td>
<td>2.78</td>
<td>7.6</td>
</tr>
<tr>
<td></td>
<td>Reptile (1)</td>
<td>1422949</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>D5 (71x)</td>
<td>SHREC</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></td>
<td>Reptile (1)</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 (193x)</td>
<td>SHREC</td>
<td>17158925</td>
<td>2947342</td>
<td>1298891</td>
<td>87495703</td>
<td>0.01</td>
<td>85.3</td>
<td>99.9</td>
<td>78.9</td>
<td>2.76</td>
<td>4.6</td>
</tr>
<tr>
<td></td>
<td>Reptile (1)</td>
<td>17158925</td>
<td>2947342</td>
<td>1298891</td>
<td>87495703</td>
<td>0.01</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 $(C_m, \sqrt{p})$ values (14, 60), (12, 60), (10, 60), (10, 55), (8, 60), (8, 55), (8, 50), (8, 45), (7, 45), (6, 45), (5, 45), respectively. The last sample point uses parameters $k = 12, d = 2, |t| = 24, C_m = 8, \sqrt{p} = 45$.

Table 4. Quality of ambiguous base correction using Reptile

<table>
<thead>
<tr>
<th>Data N</th>
<th>Accuracy (%)</th>
<th>Sensitivity (%)</th>
<th>Specificity (%)</th>
<th>Gain (%)</th>
<th>EBA (%)</th>
<th>Gain (%)</th>
<th>EBA (%)</th>
</tr>
</thead>
<tbody>
<tr>
<td>D2</td>
<td>A 99.98</td>
<td>66.4</td>
<td>100</td>
<td>63.7</td>
<td>0.01</td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>C 100</td>
<td>66.4</td>
<td>100</td>
<td>63.8</td>
<td>0.01</td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>G 100</td>
<td>66.4</td>
<td>100</td>
<td>63.7</td>
<td>0.01</td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>T 100</td>
<td>66.3</td>
<td>100</td>
<td>63.7</td>
<td>0.01</td>
<td></td>
<td></td>
</tr>
<tr>
<td>D6</td>
<td>A 99.99</td>
<td>85.1</td>
<td>99.8</td>
<td>78.5</td>
<td>0.01</td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>C 99.99</td>
<td>85.2</td>
<td>99.8</td>
<td>78.9</td>
<td>0.009</td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>G 100</td>
<td>85.3</td>
<td>99.8</td>
<td>78.5</td>
<td>0.01</td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>T 99.95</td>
<td>85.3</td>
<td>99.8</td>
<td>78.5</td>
<td>0.009</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

(14x) Reptile (1) 1169256 556586 44959 371187471 0.009 67.8 99.9 65.2 0.35 0.84
(15x) Reptile (1) 1315277 410565 91205 371141225 0.042 76.2 99.9 70.9 1.23 0.84
(172.5x) Reptile (1) 7138883 2361813 1138666 610144638 0.013 75.1 99.8 63.2 1.66 2.2
(193x) Reptile (1) 17158925 2947342 1298891 87495703 0.01 85.3 99.9 78.9 2.76 4.6

(14x) Reptile (2) 1315277 410565 91205 371141225 0.042 76.2 99.9 70.9 1.23 0.84
(15x) Reptile (2) 1315277 410565 91205 371141225 0.042 76.2 99.9 70.9 1.23 0.84
(172.5x) Reptile (2) 7138883 2361813 1138666 610144638 0.013 75.1 99.8 63.2 1.66 2.2
(193x) Reptile (2) 17158925 2947342 1298891 87495703 0.01 85.3 99.9 78.9 2.76 4.6

Table 4. Quality of ambiguous base correction using Reptile

<table>
<thead>
<tr>
<th>Data N</th>
<th>Accuracy (%)</th>
<th>Sensitivity (%)</th>
<th>Specificity (%)</th>
<th>Gain (%)</th>
<th>EBA (%)</th>
<th>Gain (%)</th>
<th>EBA (%)</th>
</tr>
</thead>
<tbody>
<tr>
<td>D2</td>
<td>A 99.98</td>
<td>66.4</td>
<td>100</td>
<td>63.7</td>
<td>0.01</td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>C 100</td>
<td>66.4</td>
<td>100</td>
<td>63.8</td>
<td>0.01</td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>G 100</td>
<td>66.4</td>
<td>100</td>
<td>63.7</td>
<td>0.01</td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>T 100</td>
<td>66.3</td>
<td>100</td>
<td>63.7</td>
<td>0.01</td>
<td></td>
<td></td>
</tr>
<tr>
<td>D6</td>
<td>A 99.99</td>
<td>85.1</td>
<td>99.8</td>
<td>78.5</td>
<td>0.01</td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>C 99.99</td>
<td>85.2</td>
<td>99.8</td>
<td>78.9</td>
<td>0.009</td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>G 100</td>
<td>85.3</td>
<td>99.8</td>
<td>78.5</td>
<td>0.01</td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>T 99.95</td>
<td>85.3</td>
<td>99.8</td>
<td>78.5</td>
<td>0.009</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
The proposed error correction algorithm is conservative because it avoids changing bases unless there is a compelling under-
representation of a tile compared with its $d$-mutant tiles. Actual errors in read $r$ cannot be corrected if $r$ occurs in a very low coverage region of the genome or there exist multiple candidate $d$-mutant tiles, probably because of genome repetition. On the other hand, a tile may be miscorrected if it contains a minor variant of a highly repetitive element in the genome or it traverses a low coverage region that is similar to other regions with normal coverage. Our method is not unique in being challenged by non-uniform coverage on repetitive genomes. Error correction for highly repetitive genomes is essential for successfully assembling larger eukaryotic genomes but none of the existing methods successfully addresses this problem, including Reptile.

Short-read mapping provides a reasonable method to evaluate error correction methods in well-assembled, low repetition genomes. Nevertheless, it is not possible to unambiguously determine all errors. There are natural polymorphisms among bacterial lines, and some presumed polymorphisms may be unrecognized assembly errors. Furthermore, the mapping software chooses among alternative mappings by invoking parsimony, but there is some chance that the true number of errors is less than the minimum. Lastly, mapping software cannot map reads that contain more than a constant number of substitutions, typically just two, with full sensitivity, although we considered 5 here and tested as many as a constant number of substitutions, typically just two, with full sensitivity, although we considered 5 here and tested as many as 15 with similar results. Despite these limitations, we believe that most errors are correctly identified, and this approach can provide a fair comparison of error correction methods.

We and others (Smith et al., 2008) have found that sequence quality scores provide valuable information. Our use of quality scores probably helped us account for the error patterns in next-generation sequencing data (Dohm et al., 2008) without explicitly modeling them. However, it has been observed (Dohm et al., 2008) that high quality scores may be too optimistic and low quality scores too pessimistic in estimating sequencing errors in Solexa data. Since quality scores may not be precise measures of misread probabilities, the current version of Reptile uses quality score information in a very simple manner, but can be modified to make more sophisticated use of quality scores if warranted. Finally, although quality scores are needed to run Reptile, it can be run effectively without scores by setting all quality scores and the threshold $Q_k$ to the same value.

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 kmers in order to address error correction in much longer reads.

**Funding:** This research is funded in part by the Iowa State University Plant Sciences Institute Innovative Research Grants program.