Pluribus—Exploring the Limits of Error Correction Using a Suffix Tree

Daniel Savel, Thomas LaFramboise, Ananth Grama, and Mehmet Koyutürk

Abstract—Next generation sequencing technologies enable efficient and cost-effective genome sequencing. However, sequencing errors increase the complexity of the *de novo* assembly process, and reduce the quality of the assembled sequences. Many error correction techniques utilizing substring frequencies have been developed to mitigate this effect. In this paper, we present a novel and effective method called Pluribus, for correcting sequencing errors using a generalized suffix trie. Pluribus utilizes multiple manifestations of an error in the trie to accurately identify errors and suggest corrections. We show that Pluribus produces the least number of false positives across a diverse set of real sequencing datasets when compared to other methods. Furthermore, Pluribus can be used in conjunction with other contemporary error correction methods to achieve higher levels of accuracy than either tool alone. These increases in error correction accuracy are also realized in the quality of the contigs that are generated during assembly. We explore, in-depth, the behavior of Pluribus, to explain the observed improvement in accuracy and assembly performance. Pluribus is freely available at http://compbio.case.edu/pluribus/.

Index Terms—Biology and genetics, trees

1 Introduction and Background

Next Generation Sequencing (NGS) technologies have replaced Sanger sequencing as the *de facto* standard. This shift can be attributed to the orders of magnitude increase in throughput and reduction in per-base sequencing cost of NGS. These desirable characteristics, however, come at the cost of shorter reads and increased error rates. In Sanger sequencing, error rates could be as low as one miscalled base in 100,000 bases sequenced. NGS technologies, on the other hand, are prone to miscalling bases at a rate that is orders of magnitude greater—as high as 1 in 100. With error rates this high and average read lengths exceeding 100 bp the expected number of miscalled bases in a read exceeds one, implying that almost every read in a sequencing run will contain at least one error [1].

There are two main uses of the reads generated by NGS technologies: (i) resequencing or mapping, where the reads are aligned to a reference genome for further analysis, and (ii) *de novo* assembly, where an unknown genome is constructed entirely from the reads. In both of these cases, sequencing errors significantly increase the complexity of operations. When performing an alignment, sequencing errors can be handled by accounting for mismatches and short gaps [2]. However, it is difficult to ascertain whether these are due to sequencing errors or true variants in the underlying genome. During *de novo* assembly, sequencing errors interfere with the identification of overlaps between reads. In this case, sequencing errors in reads that bridge two contigs may cause the contigs to stay disjoint; alernately, they may induce spurious overlaps [3]. For these reasons, it is important to discard or correct sequencing errors prior to assembly.

1.1 Error Correction

In order to identify and correct errors in sequencing data, it is necessary to differentiate true genomic variants from sequencing errors. For *de novo* assembly, there is no ground truth to compare the data against. However, if there is sufficient coverage, i.e., if the expected number of reads that cover any given location on the genome is sufficiently large, spectral alignment can be used. Spectral alignment works by approximating the set of substrings that would exist in the reference genome as the set of substrings in the read set that appear in at least a certain number of reads. Subsequently, the reads are aligned to this set of substrings to identify sequencing errors [4]. The problem of *error identification* then translates to the computational problem of identifying substrings with low frequency, since these are indicative of sequencing errors. The *error correction* problem, on the other hand, is one of determining the modifications that will transform erroneous substrings to high frequency substrings, which, in principle, correspond to true genomic sequences.

1.2 Review of Existing Methods

Building on the spectral alignment approach to error correction, several algorithms and tools have been proposed. Types of errors that are commonly encountered in next generation sequencing are illustrated in Fig. 1. Based on this categorization, the tools can be classified into two groups:

- *D. Savel and M. Koyutürk are with the Department of Electrical Engineering and Computer Science, Case Western Reserve University, 10900 Euclid Avenue, Cleveland, OH 44106. E-mail: {dan.savel, mehmet.koyuturk}@case.edu.*
- *T. LaFramboise is with the Department of Genetics and Genome Sciences, Case Western Reserve University, 10900 Euclid Avenue, Cleveland, OH 44106. E-mail: thomas.laframboise@case.edu.*
- *A. Grama is with the Department of Computer Science, Purdue University, 305 N. University Street, West Lafayette, IN 47907. E-mail: ayg@cs.purdue.edu.*

Manuscript received 2 June 2015; revised 30 Apr. 2016; accepted 9 May 2016. Date of publication 29 June 2016; date of current version 6 Dec. 2017.
For information on obtaining reprints of this article, please send e-mail to: reprints@ieee.org, and reference the Digital Object Identifier below.
Digital Object Identifier no. 10.1109/TCBB.2016.2586060

1545-5963 © 2016 IEEE. Translations and content mining are permitted for academic research only. Personal use is also permitted, but republication/redistribution requires IEEE permission. See http://www.ieee.org/publications_standards/publications/rights/index.html for more information.

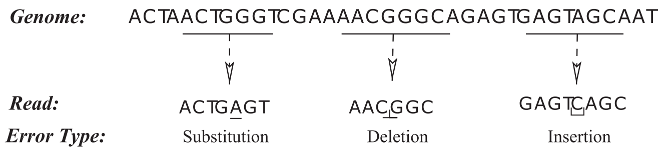

Fig. 1. Illustration of the types of errors that can be encountered in sequencing.

tools that can correct only *substitution* type errors and tools that can correct both substitution and *indel* type errors. Since the most popular sequencing platform today, Illumina, has higher rates of substitution errors and lower rates of indel errors, many of the recent algorithms focus exclusively on substitution errors. Such tools include QUAKE [5], SHREC [6], HITEC [7], MUSKET [8], RACER [9], and REPTILE [10]. Tools that target indels are also available; for example, HYBRID-SHREC [11] provides an extension of SHREC that can correct errors in reads that could have come from a mixture of hardware platforms, so it has the ability to correct both substitution and indel type errors.

Differences between spectral alignment based error correction tools arise from the underlying data structures used to store and analyze substring frequency. Three data structures are commonly used for this purpose: (i) K-mer based lookup tables, (ii) suffix tries, and (iii) suffix arrays. K-mer based lookup tables usually count the frequencies of all substrings of length K (K-mers) for a fixed K, and identify as errors, sets of overlapping K-mers with low frequency [5], [10]. For these tools, K is a user-defined parameter and has a significant effect on the performance of error correction. A small value of K tends to result in a set of all high frequency K-mers, and a large value of K tends to result in a set of all low frequency K-mers. Tools typically provide default parameters, or a simple estimator, to determine K [5]. Suffix trie based methods, on the other hand, remove the dependency on a fixed parameter (K) by organizing all suffixes of all reads into a trie structure, and identify as errors, low-frequency nodes with high-frequency siblings [6], [11]. A third data structure, suffix array, is used as a compromise between K-mer based lookup tables and suffix tries. In a suffix array, all the suffixes of the reads are still considered but they are organized into a flat data structure and lexicographically sorted [7].

Instead of using the spectral alignment, it is also possible to perform multiple sequence alignment (MSA) for error correction. CORAL [12] is an error correction tool that uses MSA to identify and correct sequencing errors. CORAL still uses K-mers, but they are used to index reads rather than to identify the errors themselves. Once K-mers are indexed, CORAL generates consensus sequences from overlapping reads using MSA, and identifies the gaps and mismatches to be translated into corrections in aligned reads. When compared to spectral alignment based methods, MSA-based methods offer the benefit of allowing inexact matches during alignment at the cost of increased running time.

1.3 Contribution of Our Work

Owing to the space requirements of the suffix trie data structure, many suffix trie based algorithms attempt to identify errors using partially constructed tries. This entails identification of errors in a *trie-driven* manner, i.e., each error is identified based on a single node of the trie. The decision on how to correct this error is based only on the siblings of that node. However, a single erroneous base in a read is incident on a number of suffixes equal to its index in the read, or a number of suffixes equal to the length of the read when taking into account both forward and reverse complement directions. Each of these suffixes that are incident on the error can manifest themselves in the trie as a low frequency node, and these correlated manifestations can be used together to accurately identify how to correct that error.

We propose PLURIBUS, a *read-driven* algorithm that considers all manifestations of an error in guiding the correction process. Since PLURIBUS uses multiple manifestations of a read to identify errors and suggest corrections, it is expected to be more precise in detecting errors and more accurate in suggesting corrections. We compare the performance of PLURIBUS and other methods in detecting and correcting sequencing errors across a diverse set of genomes and as a function of coverage. We also assess the impact of error correction on the performance of *de novo* assembly by comparing the performance of a state-of-the-art assembler, Velvet using uncorrected reads as well as the corrected reads. Finally, we explore in-depth the behavior of PLURIBUS to explain our accuracy and assembly results. Our results show that utilizing multiple suffixes to correct errors results in a precise method for correcting sequencing errors. We show that PLURIBUS can be used in conjunction with other tools to improve the accuracy of error correction, which results in a significant improvement in the quality of the contigs generated in assembly.

In the next section, we formalize notation and describe, in depth, the process of error correction via a suffix trie. We then explore the correction performance of several tools including an analysis of the behavior of suffix trie methods. Finally, a comprehensive analysis of suffix trie based error correction is presented.

2 METHODS

We first introduce our notation and formally define the error correction problem. We then explain how a suffix trie based data structure can be used to detect and correct sequencing errors, as is done by existing methods. We then discuss shortcomings of these methods, and present our method, PLURIBUS, for suffix trie based error correction utilizing multiple suffixes simultaneously.

2.1 Notation and Problem Formulation

The input to the sequencing error correction problem is a set $\mathcal{R}$ of $n = |\mathcal{R}|$ short reads from a genome G of length m (the genome length is not necessarily known). The reads are not of identical length, but we can safely assume that the average read length, ℓ, is known. For such a set of reads, the coverage c is defined as the average number of reads that contain a given base, i.e., $c = n\ell/m$. The coverage of a sequencing run depends on multiple factors including the sequencing platform, the resources available, and the length of the genome being sequenced. Coverage levels in sequencing runs intended for *de novo* assembly typically range from $16\times$ to more than $200\times$ [13].

Because of the technological limitations of sequencing platforms, reads in a sequencing run contain errors, i.e., one

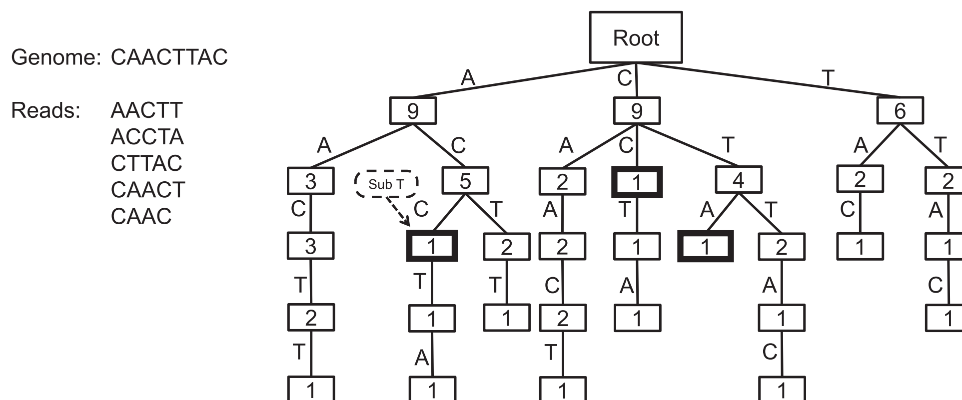

Fig. 2. **Use of suffix tries for the detection and correction of sequencing errors:** A simple hypothetical genome and a set of five reads from this genome are shown on the left. The second read contains a substitution error, where a T is substituted with a C. The suffix trie that indexes all suffixes of these reads is shown on the right. Each node in the trie represents the string that is obtained by reading along the path from the root to that node. The number at each node corresponds to the frequency of the respective string as a substring of the reads in the read set. The nodes highlighted in bold are those that are low frequency, which also have siblings that are high frequency (assuming a threshold of 1). If the trie is traversed in lexicographical order, the correction highlighted by dashed lines will be the first correction first suggested.

or more bases in a read may not be identical to those at their corresponding position in G. The objective of sequencing error correction is to identify and correct such errors in reads. Various types of errors encountered in sequencing are illustrated in Fig. 1. The first illustrated error type is the substitution error. In this type of error, a nucleotide in a read is substituted with another nucleotide. Errors can occur in the form of insertions and deletions as well, where one or more nucleotides are inserted within a read, or one or more nucleotides are deleted from within a read. These types of sequencing errors are commonly referred to as indels.

The output of error correction is a set of suggested corrections, where for each read $\Delta_i^{(\alpha)}$ denotes a suggested correction. The type of correction that is to be applied is noted as α and the position in the read to apply that correction is i. The possible values for α are considered to be "Insert" followed by a base, "Substitute" followed by a base, and "Delete".

2.2 Suffix Trie-Based Error Correction

Correction of sequencing errors typically relies on spectral alignment, i.e., the frequencies of the substrings of the reads. This is based on the premise that with sufficient level of coverage, substrings that come from an actual genomic sequence will have relatively high frequencies, whereas substrings that contain an error will have relatively low frequencies. Therefore, one can identify sequencing errors by computing the frequencies of all substrings of the reads in $\mathcal{R}$ and identifying substrings that have frequency lower than τ, a threshold between high and low frequency that is either user-specified or determined analytically.

Suffix tries provide a useful data structure for spectral alignment, since they can be used to track the frequencies of substrings of any length by indexing all suffixes of all reads in $\mathcal{R}$. The trie is constructed in such a way that the paths in the trie represent the substrings of the reads in $\mathcal{R}$, and the individual nodes represent the frequency of the substring defined by the path from the root of the trie to that node. Since the sequencing machinery cannot distinguish between the two strands of a DNA sequence, the reverse complement of all suffixes are also indexed. Once all suffixes are indexed, errors manifest themselves as low-frequency subtrees with roots that have high-frequency siblings in the trie.

Formally, the trie is composed of a set of $Node$ objects, where each $Node$ has a single parent node, $Node.Parent$, and has a set of children nodes, $Node.Children$. Each $Node$ also contains the two values $Node.Character$ and $Node.Frequency$, which respectively store the character represented by this node and the frequency of the substring that ends at this node. Potential errors in this trie are represented by pairs of nodes, $(Node_x, Node_y)$ such that

$$\begin{gathered} Node_x.Parent = Node_y.Parent \wedge \\ Node_x.Frequency \leq \tau \ \wedge \ Node_y.Frequency > \tau. \end{gathered} \tag{1}$$

Besides revealing potential errors, these $Node$ pairs suggest possible corrections that could be applied to the read(s) that contain the substring represented by $Node_x$. We explain the correction process below in the paragraph titled "Verification Levels". Existing algorithms that use suffix tries for error correction include SHREC [6] and HYBRID-SHREC [11]. HiTEC [7] also uses suffixes in a similar manner, to correct sequencing errors, however, the suffixes are stored in a suffix array as opposed to a trie.

An example illustrating suffix trie based error correction is shown in Fig. 2. This examples shows the trie constructed from a simple hypothetical read set. In this read set, there are three suffixes with low frequency (1 in this example), for which there are other suffixes in the read set that differ by a single base and have high frequency (2 or more in this example). These three suffixes are highlighted in bold in the figure. Specifically, the node that is highlighted in bold is the first character for which the substring becomes low frequency. If the trie is traversed in lexicographical order, the leftmost highlighted node will be identified first. If the first character of this node is changed to a 'T' the remaining characters of the highlighted node will match the characters below the sibling node. This pair of nodes suggests the correction $\Delta_3^{(SubT)}$, i.e., the third base of the second read should be a 'T' not a 'C'. This action will accurately correct the single error in the read set. The suggested correction is represented in Fig. 2 by a node with a dashed outline and a dashed arrow pointing at the node that suggested that correction.

Construction Levels (J to K). The generalized suffix trie tracks the frequency of substrings of all lengths, however not all lengths of substrings actually provide utility for error correction. As noted in the discussion of K-mer based algorithms, when counting the frequency of substrings the chosen length will have a significant impact on the performance of the algorithm. When the chosen length is too small, the total number of possible substrings becomes saturated and all substrings tend to have high frequency. If the chosen length is too large, all the substrings will tend to have low frequency because of errors. In both of these cases, the substrings are unable to distinguish between true sequences and errors.

This observation also applies to suffix trie based error correction. Namely, there is a range of suffix lengths that are large enough such that not all suffixes are high frequency and small enough such that not all of the suffixes are low frequency. Therefore, it is possible to only construct a specific band or range of suffixes to reduce the time and space requirements of trie construction without compromising the accuracy of error detection and correction. We denote the upper and lower bounds of this range as J (length of the shortest possible suffix in the trie) and K (length of the longest possible suffix in the trie). For existing suffix trie-based algorithms, J and K are user-defined parameters. In the following discussion, we use the terms "level" (i.e., depth of a node in the suffix trie) and "length" (of a substring) interchangeably since each node of the suffix trie represents a specific substring and its depth in the trie is equal to the length of the substring.

Not all levels that are constructed are utilized for the same purpose. Instead, some of them can be used used to detect possible sequencing errors, while others can be used to verify a suggested correction.

Detection Levels (D). Recall that the constructed suffix trie contains $K - J + 1$ levels, where J and K respectively denote the length of the shortest and longest substrings used in the construction of the trie. While it is possible to use all these levels to detect errors and identify suggested corrections, the quality of the corrections suggested by each level is not identical. The expected frequency of nodes decreases for each successive level of the trie, so the fraction of invalid suggestions may increase. Furthermore, as suggestions are made from successively lower levels of the tree, there are fewer available characters to verify them. For this reason, existing suffix trie based correction algorithms detect errors using a restricted set of levels in the suffix trie. We refer to these levels of the trie as detection levels and denote the number of detection levels as D. The implemenation of SHREC [http://sourceforge.net/projects/shrec-ec/] and by extension HYBRID-SHREC use only a single detection level ($D = 1$). The frequencies of substrings shorter than the length of the substrings in the detection level are not considered. For this reason, we assume that level J is always the first detection level utilized by the algorithm.

Verification Levels (V). When the number of constructed trie levels is larger than the number of levels used for detection, the remaining levels of the trie can be used as verification levels to further filter suggested corrections. We denote the number of verification levels by V. Since both SHREC and HYBRID-SHREC use only a single level of the trie as a detection level, they use the remaining constructed levels as verification levels (i.e., $V = K - J$ for these algorithms) (the default setting for SHREC is $V = 4$). In order for a correction $\Delta_i^{(\alpha)}$ to be verified, the $(K - J)$ characters that follow position i in the read must match all high frequency nodes in the subtree rooted at the high frequency neighbor node that corresponds to α.

A substring in the tree is represented by a path, so the verification process can be formally described in terms of these paths. Let P_{read} be the path of nodes that represents the substring of length K that $\Delta_i^{(\alpha)}$ is suggesting to modify. Let P_{test} be the path of nodes that represents what the substring will become if $\Delta_i^{(\alpha)}$ was applied. Verification of a substring means that a correction is only applied if the following equation evaluates to true:

$$\forall_i (J < i \le K \ \wedge \ P_{test}[i].Frequency > \tau \\ \wedge \ P_{test}[i].Character = P_{read}[i].Character). \tag{2}$$

Since the nodes that suggest $\Delta_i^{(\alpha)}$ are siblings, the $J - 1$ nodes of P_{read} are already known to overlap and have high frequency. Furthermore, since a high frequency node is used to identify the correction, we know that the first non-overlapping node of P_{read} also has high frequency. Therefore, to verify a suggestion, only the last $(K - J)$ nodes need to be checked. The remaining nodes can then be checked by traversing through the subtree that is rooted at the high frequency sibling, $P_{test}[J]$. Equation (2) specifically refers to checking substitution corrections. For insertion type corrections $P_{test}[i]$ is compared against $P_{read}[i-1]$, while for deletion type corrections $P_{test}[i]$ is compared against $P_{read}[i+1]$.

2.3 Using Multiple Suffixes for Correction

The general approach to using a suffix trie for error correction, as implemented by SHREC and HYBRID-SHREC is to detect errors on the trie, and use references back to $\mathcal{R}$ to correct the detected error. In this approach, the correction process only has a single substring in its scope at any given time. Therefore, while working from the trie, error correction decisions are made using the information from a single substring, and other manifestations of an error are not taken into account. This "trie-driven" approach offers an important advantage: Since correction is being performed based on the information from a single substring, only a fraction of the suffix trie needs to be in memory at any given time.

The trie-driven approach also has an important drawback: Not all suggested corrections are valid. They may be ambiguous, since a single manifestation may suggest multiple corrections, especially if a small number of verification characters are used. In fact, some may potentially be spurious, and not related to an actual sequencing error. These caveats are exacerbated when the error profile contains indels as well as substitution type errors, since there is a larger number of possible corrections that could be applied to a substring when indels are also considered. This larger number of possibilities increases the likelihood of finding a possible modification to a read that exists just by chance rather than an actual sequencing error.

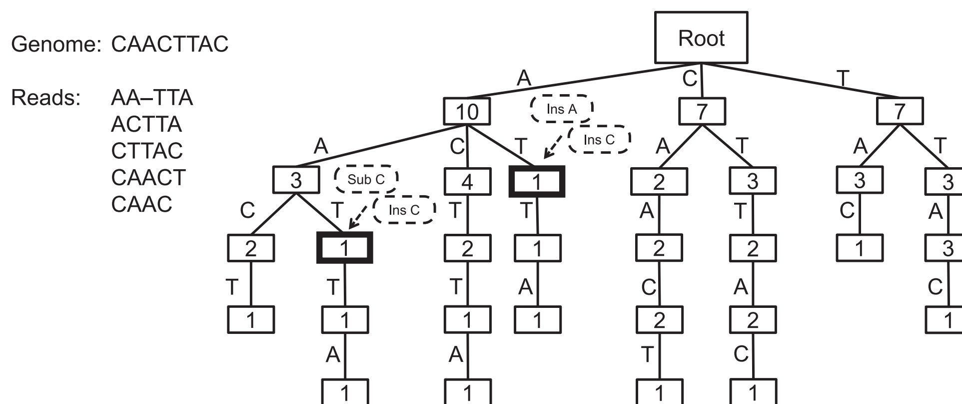

Fig. 3. **Illustration of the shortcomings of trie-driven error detection:** The generalized suffix trie for a set of five reads with a single deletion error made from a simple hypothetical genome. The nodes highlighted in bold are those that are low frequency, which also have siblings that are high frequency (assuming a threshold of 1). The items highlighted by dashed outlines are all of the corrections suggested by the trie with dashed lines denoting the specific node that suggested those corrections. Note that both nodes each suggest two different corrections, so a trie-driven method would only be able to guess which one to apply. However, looking at the set of suggestions as a whole reveals that there is a common suggestion between the nodes, and a read-driven method would be able to identify the whole set of suggestions before applying a correction.

The example illustrated in Fig. 3 shows the trie for a read set that contains a single deletion error. In this example, there are two nodes such that $Node.Frequency \leq \tau$ which also have sibling nodes where $Node.Frequency > \tau$. These two instances are highlighted in bold. The leftmost highlighted node has a single high frequency sibling. This pair of nodes actually suggests two corrections, $\Delta_3^{(SubC)}$ and $\Delta_3^{(InsC)}$, and these are represented by the two leftmost highlighted items in Fig. 3. One suggestion is valid, while the other is a spurious suggestion caused by the use of a single verification character. The second highlighted node has two high frequency siblings which also suggests two corrections $\Delta_3^{(InsC)}$ and $\Delta_3^{(InsA)}$, and this pair of suggestions is represented by the two rightmost highlighted items in Fig. 3. Again only one suggestion is valid, and the other is actually an artifact of a short repeat. A trie-driven method will only discover one of these nodes at a time, and using the information at hand the method only has a 50 percent chance of picking the right correction.

Algorithm 1. Pluribus

Input: $\triangleright$ $\mathcal{R}$: Set of reads to be corrected
Output: $\triangleright$ $\mathcal{R}'$: Set of corrected reads
1. $\mathcal{T} \leftarrow$ generalized suffix trie constructed from $\mathcal{R}$
2. $\mathcal{R}' \leftarrow \emptyset$
3. **for** each read $r \in \mathcal{R}$ **do**
4. $C \leftarrow \emptyset$
5. **for** each suffix $s \in r$ **do**
6. Query $\mathcal{T}$ for s
7. **if** s is incident on a low-frequency node **then**
8. **for** each possible correction **do**
9. **if** correction allows s to match all high-frequency nodes **then**
10. Add correction to C
11. **end if**
12. **end for**
13. **end if**
14. **end for**
15. MostFrequentCorrection $\leftarrow$ most frequent item in C
16. Apply MostFrequentCorrection to r
17. $\mathcal{R}' \leftarrow \mathcal{R}' \cup r$
18. **end for**

2.4 Correcting Errors with Pluribus

In order to address the shortcomings of trie-driven error correction, we propose, Pluribus, a read-driven algorithm for error correction. Pluribus considers each read one by one, and refers to the trie to find all low frequency substrings incident on the read. This "read-driven" approach used by Pluribus examines all suffixes of each individual read and identifies all the corrections that could be applied to the read. These suggested corrections are not always concordant. Therefore, in order to handle discrepancies between suggested corrections, Pluribus utilizes a voting scheme. This voting scheme considers all suggested corrections for each read, and realizes the most frequently suggested correction for that read. This approach is described in detail in Algorithm 1.

To be more precise, we denote the set of suggested corrections as C. As shown in Algorithm 1, C is defined for each read and all possible suggestions are added into it as the read is queried against the constructed trie. Defining $Count(C, \Delta_i^{(\alpha)})$ as the number of times a specific correction is suggested during this phase, Pluribus chooses the correction to be applied as

$$\arg\max_{\Delta_i^{(\alpha)}} \; Count(C, \Delta_i^{(\alpha)}). \tag{3}$$

In this manner, Pluribus guarantees that the modification performed on a read is consistent for any arbitrary traversal of the underlying data structure, and for any arbitrary order of the reads in $\mathcal{R}$.

Referring back to the deletion example in Fig. 3, Pluribus would identify both of the highlighted nodes and would identify all of the suggested corrections. The set of suggested corrections would be $\{\Delta_3^{(SubC)}, \Delta_3^{(InsC)}, \Delta_3^{(InsA)}, \Delta_3^{(InsC)}\}$. Inserting a 'C' after the second base is suggested multiple times; it is the most frequently suggested item and the only item suggested more than once. This most frequent correction is the one that Pluribus chooses to apply to the read, and it is the only valid correction in the set.

Applying Correction in Multiple Rounds. Depending on the error rate of the sequencer, it is possible that an error can contain multiple errors. This is indeed the case for the

Illumina platform. While simultaneous identification and correction of multiple errors on a read is desirable, it is computationally prohibitive since the space of possible corrections grows exponentially with the number of errors on a read. An approximation to simultaneous correction would be to use a fully dynamic suffix trie, i.e., a trie is modified everytime a read is modified to properly represent the corrected read set. However, the construction and utilization of such a tree is also not feasible for the current ranges of genome length and coverage.

PLURIBUS takes a more modest, yet reliable approach to this problem: The whole process is performed in rounds where up to a single correction is applied to each read before the trie is rebuilt using the modified reads. Note that the corrections applied to the read set can improve or enable the correction of additional errors in two different ways: First, correcting errors in a read can make it easier to identify corrections for other errors in that read. Therefore, correction in multiple rounds provides a tractable method of using information gained by correcting reads throughout the read set to improve correction of other reads.

2.5 Complexity Analysis of PLURIBUS

The main component of the runtime and space requirements of the algorithm is the construction and querying of the suffix trie. The algorithm essentially builds a generalized suffix trie, then walks through the trie again during its search for suggested corrections. At a first glance, the algorithm may imply a runtime and space complexity of $O(n\ell^2)$. However, as described above, only a small subset of the trie is actually constructed. The parameters J and K dictate the first and last levels of the trie to be constructed. By limiting the length of the longest substring stored in the trie to be K, the complexity of a naive trie construction becomes $O(n\ell K)$. Furthermore, since the length of the shortest substrings stored in the trie is J the complexity becomes $O(n\ell(K-J))$ as long as nodes representing substrings of length J can be accessed in constant time (e.g., using a hash table). Since the value of $K-J$ tends to be small (around 4 or 5), the complexity of the algorithm can be considered as $O(n\ell)$ for practical applications.

2.6 Reverse Complement Considerations

In sequencing, reads can come from both strands of DNA. Therefore, both the read and its reverse complement need to be considered in spectral alignment. In practice, when storing substring frequencies, both the read and its reverse complement are stored in the data structure that is being used to count frequencies. Corrections can then be identified in either direction of a read. Taking the reverse complements of the reads into account does not significantly impact the complexity of correction algorithms, since in the worst case, the number of substrings stored is doubled. PLURIBUS handles reverse complements by adding suggested corrections from both directions into its set of suggestions for a read, C. For practical convenience, when corrections suggested by the reverse complement are added into C, they need to be transformed to use the forward direction as the frame of reference for the position of the correction and the nucleotide to change into (if applicable).

TABLE 1
Test Datasets

Accession	Organism	Coverage	Genome Length
SRR519926	E. Coli	43×	4.64 Mbp
SRR1203044	S. Enterica	89×	4.89 Mbp
SRR1206093	S. Enterica	97×	4.89 Mbp
SRR1119292	P. Synringae	105×	6.09 Mbp

3 RESULTS

In this section, we compare PLURIBUS against existing error correction methods using real sequencing datasets. We first examine the accuracy of correction of different methods using the tools from a recent correction survey[14]. Then, we use QUAST[15] to compare the assembly performance of the datasets corrected by the suite of correction methods. Finally, we present an in-depth analysis of the behavior of PLURIBUS to explain our results.

3.1 Experimental Setup

We compare the performance of PLURIBUS against the state-of-the-art error correction methods MUSKET[8] and RACER [9]. These tools were chosen based on their overall performance as presented in [14]. We have chosen four Illumina MiSeq real sequencing datasets for our test data. One dataset is for Escherichia Coli, two are for Salmonella Enterica, and the last one is for Pseudomonas Syringae. Details on the four datasets is presented in Table 1.

The read lengths of the datasets range up to 251 bp. In our earlier experiments on other data sets [16], we found that performing three rounds of correction with PLURIBUS was sufficient to correct reads of around 100 bp in length. For this reason, we perform six rounds of correction with PLURIBUS on these four datasets.

3.2 Correction Accuracy

In the survey of error correction methods [14], several metrics that reflect the accuracy of correction were introduced. We use the tools READSEARCH and KMERSEARCH from [14], which respectively compute the performance criteria READDEPTHGAIN and KMERDEPTHGAIN. These criteria are described in detail by Molnar et al. [14]. Briefly, these criteria quantify the trade-off between precision and recall specifically in the context of error correction, i.e., it quantifies the gain achieved by detecting and correcting errors at the cost of making spurious corrections as well. We use these criteria to assess the correction performance of PLURIBUS. The difference between READDEPTHGAIN and KMERDEPTHGAIN arise from the unit of analysis that is considered, i.e., whether the assessment is done at the level of reads or at the level of substrings with fixed length.

Table 2 shows READDEPTHGAIN and KMERDEPTHGAIN for the correction of the test datasets listed in Table 1. A partial breakdown of the components used to calculate KMERDEPTHGAIN is also presented in Table 2 to provide a more in-depth comparison of accuracy. The partial breakdown is comprised of the metrics “Specificity”, “Sensitivity”, “True Positives”, and “False Positives”, which are all reported along with KMERDEPTHGAIN by KMERSEARCH. The best performance for each metric on each dataset highlighted in bold.

TABLE 2
Error Correction Accuracy

Dataset	Tool	ReadDepthGain	KmerDepthGain	Specificity	Sensitivity	True Positives	False Positives
SRR519926	Musket	23.09	30.04	99.994	30.05	27,593,420	5,041
	RACER	26.36	37.75	99.942	37.81	34,718,685	54,226
	Pluribus	9.72	19.50	**99.997**	19.50	17,909,063	**2,427**
	PRCor	**29.48**	**40.95**	99.949	**41.00**	**37,646,364**	47,797
	RPCor	28.65	39.30	99.944	39.36	36,140,285	52,035
SRR1203044	Musket	6.23	16.66	93.468	**34.13**	**37,080,221**	18,982,092
	RACER	13.54	19.73	99.965	19.83	21,539,795	105,587
	Pluribus	9.41	13.32	**99.986**	13.36	14,515,335	**40,096**
	PRCor	13.35	**20.32**	99.964	20.41	22,178,344	105,878
	RPCor	**13.55**	20.12	99.965	20.22	21,962,719	100,526
SRR1206093	Musket	26.72	45.16	95.506	**70.22**	**46,410,317**	16,557,834
	RACER	76.29	60.05	99.991	60.10	39,727,320	33,734
	Pluribus	50.85	39.01	**99.997**	39.03	25,796,452	**12,143**
	PRCor	**77.88**	**62.70**	99.988	62.76	41,484,750	43,758
	RPCor	77.86	61.51	99.991	61.56	40,687,947	33,539
SRR1119292	Musket	10.03	20.29	89.152	**42.80**	**82,244,121**	43,254,869
	RACER	16.95	21.87	99.951	21.97	42,215,514	196,754
	Pluribus	5.54	13.28	**99.975**	13.34	25,625,061	**99,448**
	PRCor	**17.24**	**22.73**	99.945	22.85	43,906,650	219,401
	RPCor	17.20	22.38	99.951	22.48	43,202,401	195,539

As shown in the table, across the four datasets Pluribus has lower ReadDepthGain and KmerDepthGain compared against Musket and RACER. However, careful investigation of the partial breakdown of KmerDepthGain leads to more interesting observations. Namely, across the four datasets, Pluribus consistently achieves the highest level of precision. To be more precise, Pluribus has the highest level of reported "Specificity" as well as the lowest number of "False Positives". The breakdown also shows why Pluribus has relatively low ReadDepthGain and KmerDepthGain. Since it is designed as a conservative method that aims to identify accurate corrections, Pluribus has the lowest reported levels of "Sensitivity" and "True Positives".

This observation suggests that Pluribus can be utilized to improve the precision of other methods in error correction. As discussed in the previous section, one of the design decisions in Pluribus is that error correction is applied in over a series of rounds. Musket also performs correction in a series of stages. However, rather than using the same method to correct at each stage as in Pluribus, Musket uses increasingly aggresive methods to correct the reads. Inspired by this approach, we use Pluribus as a conservative starting point for a compatible error correction tool, RACER. Overall, RACER has the best standalone performance and Pluribus has the highest rate of precision. This observation suggests that the extremely low rate of false positives of Pluribus can be used to perform the most reliable corrections at the beginning, thereby minimizing potential negative effects that compounding mistakes in correction. For this purpose, we run systematic experiments to compare the performance of RACER against two combinations of the two algorithms: PRCor and RPCor. PRCor represents correcting data with Pluribus first then correcting the data with RACER. RPCor represents correcting data in the oppposite order, using RACER first and Pluribus second. The results of this analysis are also shown in Table 2.

As seen in Table 2, across all four datasets PRCor and RPCor both provide improved ReadDepthGain and KmerDepthGain, as compared to RACERFurthermore. PRCor outperforms RPCor in most of the instances. Namely, as expected, prior application of the more conservative correction method leads to improved overall correction performance.

The reads in the datasets SRR1203044 and SRR1206093 are both from S. enterica. Interestingly there is a large performance difference between the two datasets. According to Table 1, the coverage of the better performing dataset is slightly higher. To assess whether the differences in performance are due to the increased coverage, we perform a series of downsampling experiments. For this purpose, we artificially reduce the coverage of the datasets by randomly dropping reads from the datasets and assess the performance of three methods on these downsampled (lower coverage) data: RACER only, PRCor (Pluribus first, then RACER), and RPCor (RACER first, then Pluribus). Fig. 4 shows how coverage effects KmerDepthGain for these three methods. Each point shows the average across five randomized runs, except the rightmost points which represent the full datasets. Error bars were left off the figure since the standard deviations were negligibly small. As seen in the figure, KmerDepthGain is quite stable with decreasing coverage, until the coverage gets lower than $10\times$. For all levels of coverage, consistently PRCor outperforms RPCor, and RPCor outperforms RACER, suggesting that a combination of a conservative and an aggressive tool outperforms the application of the aggressive tool alone.

3.3 Runtime Performance

An important consideration for error correction method is its scalability, specifically, the amount of time required to perform correction. Table 3 shows the timing results for the different correction methods when run on a desktop computer with an Intel Xeon 3.4 Ghz Quad-Core Processor and

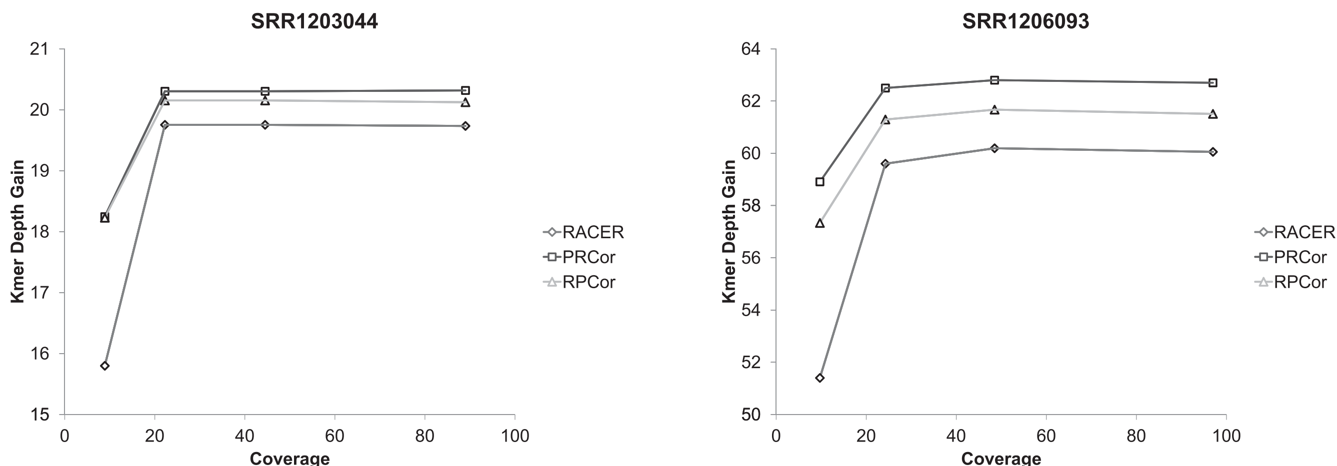

Fig. 4. Comparison of KmerDepthGain of correction for RACER and the combination of RACER and Pluribus as coverage changes for two Salmonella datasets.

16 GB of RAM. The table shows both the wall-time and the CPU-time, which takes into account the use of multi-threading. Consistently, RACER has the fastest wall-time of the methods across all datasets. However, when observing CPU-time, Pluribus requires the least amount of total CPU usage. RACER and Musket both take advantage of multi-threading and this is apparent by examining the differences between their wall-times and CPU-times.

3.4 Assembly Performance

The primary goal of performing error correction is to improve the process of *de novo* assembly. We use the Velvet assembler [3] to assemble the raw datasets as well as the datasets corrected by the suite of correction methods. Then, we use the tool QUAST[15] to assess the quality of the contigs generated by Velvet. We present the metrics NGA50, number of misassemblies, and the fraction of the reference genome that is assembled in Table 4. The best QUAST results for each metric and each dataset are bolded in the table.

For almost all metrics across all four datasets, the combinations of Pluribus and RACER have the best performance. Between PRCor and RPCor, PRCor is typically better than RPCor in assembly performance, with the E. Coli dataset being the exception. In general, the combination of the two tools increases the length of quality contigs that are generated, decreases the number of misassemblies in the contigs, and increases the fraction of the reference genome that is covered by at least one quality contig.

As mentioned above, the SRR1203044 and SRR1206093 datasets are both from the organism S. enterica. We perform the same downsampling analysis with regards to assembly performance. Fig. 5 shows how coverage effects NGA50 in the two datasets. Again, each point in the figure represents the average NGA50 across 5 runs, and the rightmost points represent the full datasets. For SRR1203044, the performance of PRCor and RPCor actually outperforms RACER on the downsampled datasets, even though they had lower NGA50 scores on the full dataset. Overall, the same trends that occured in the KmerDepthGain downsampling analysis are reflected in the assembly performance results. NGA50 stays relatively stable until coverage drops to less than $10\times$ at which point NGA50 drops dramatically for all methods.

TABLE 3
Runtimes in Seconds to Correct Each Dataset

Dataset	Tool	Wall-Time	CPU-Time
SRR519926	Musket	740.5	5820.0
	RACER	**75.5**	415.4
	Pluribus	239.3	**252.8**
SRR1203044	Musket	241.1	1907.5
	RACER	**123.6**	897.8
	Pluribus	276.8	**274.6**
SRR1206093	Musket	365.6	2891.7
	RACER	**168.3**	1235.3
	Pluribus	357.9	**372.5**
SRR1119292	Musket	471.7	3726.0
	RACER	**230.7**	1735.4
	Pluribus	496.4	**506.6**

TABLE 4
Assembly Performance via QUAST

Dataset	Tool	NGA50	Misassemblies	Fraction
SRR519926	Raw	-	11	39.43
	Musket	3,849	511	63.81
	RACER	18,519	239	73.03
	PRCor	11,131	410	70.50
	RPCor	**29,847**	**147**	**83.30**
SRR1203044	Raw	8,379	178	66.07
	Musket	32,770	156	82.97
	RACER	**72,928**	65	90.89
	PRCor	70,987	64	**91.44**
	RPCor	69,548	**63**	90.90
SRR1206093	Raw	826	123	53.28
	Musket	31,123	193	83.74
	RACER	118,646	42	93.94
	PRCor	**140,535**	**33**	**94.92**
	RPCor	90,857	45	93.99
SRR1119292	Raw	-	113	44.43
	Musket	15,853	217	71.85
	RACER	31,608	165	85.30
	PRCor	**46,819**	**101**	**86.88**
	RPCor	35,627	152	85.29

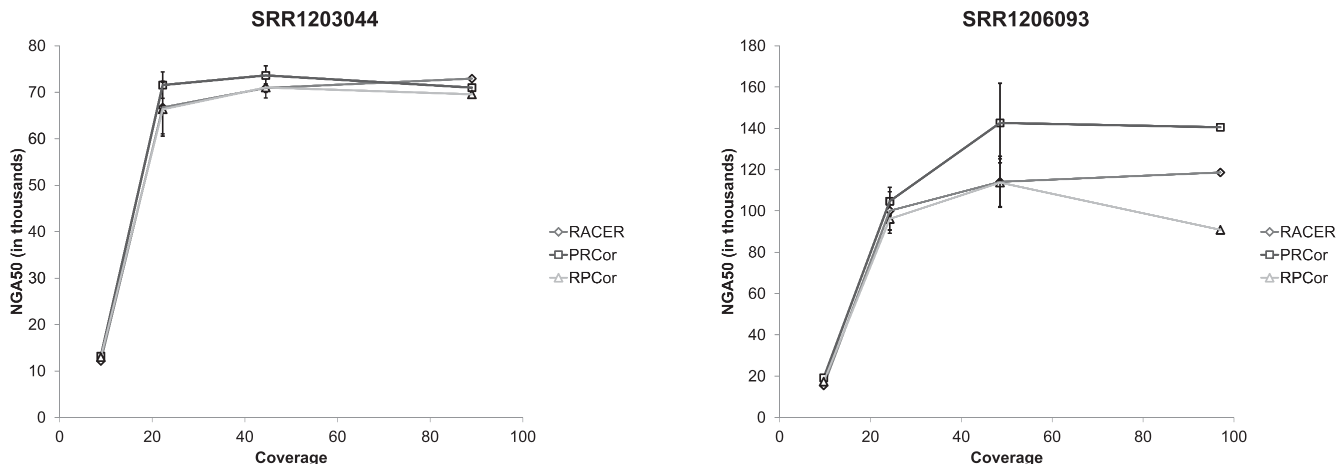

Fig. 5. Assembly performance in terms of NGA50 as coverage changes for two Salmonella datasets.

3.5 Detailed Analysis of Pluribus

As shown in the results above, Pluribus has excellent precision, but this comes at the cost of lower recall. Here, we present an in-depth analysis into the behavior of Pluribus in order to explain this behavior of Pluribus. This analysis also provides insights into the limits of suffix-tree based error correction.

3.5.1 Correctability Analysis

To identify the strengths and shortcomings of our multi-suffix approach to error correction, we classify sequencing errors in different read sets based on the ability of the method to identify valid corrections for those errors. We use simulated data since we will have the true set of corrections which serve as the basis for the classification. Besides the true corrections the classification scheme also needs the set of candidate corrections, C. We developed a modified version of Pluribus which rather than correct the sequencing errors that are detected it merely reports C for each read. These two sets of corrections are then used to classify each error according to the scheme described below.

We define five classes of errors: Undetectable, Indistinguishable, Single Suffix Correctable, Multi-Suffix Correctable, and Uncorrected and two classes for false positives Single Suffix False Positive and Multi-Suffix False Positive. All these classes are mutually exclusive (i.e., an error belongs to exactly one of these classes), except for Uncorrected and Multi-Suffix False Positive, as explained below. Each of these classes are defined as follows.

- *Undetectable* errors are sequencing errors for which there is no item in C that could correct this error.
- *Indistinguishable* errors are instances where C will contain both valid and invalid corrections for this error and the voting method would not pick one over the other.
- We consider an error to be *Single Suffix Correctable* if the error is detectable and the set of candidate corrections are either unanimous and valid or all candidate corrections are valid.
- We classify an error as *Multi-Suffix Correctable* if the candidate corrections are not unanimous and the voting scheme leads to a valid correction.
- The *Uncorrected* errors are all the errors that do not match any of the above classes. Primarily these are errors that are not corrected due to false positives.

The above set of classes are defined for an error that exists in the set of reads. We also define two classes for corrections that do not correspond to an error, i.e., false positives.

- A *Single Suffix False Positive* is a correction suggested by unanimous votes on a position where an error does not exist or the correction is not accurate for the corresponding error.
- A *Multi-Suffix False Positive* is a correction where the candidate correction that is supported by most votes for an error is not the accurate correction.

3.5.2 Simulation Setup

We use simulated data for our computational experiments, so that the “ground truth” for each read is available. For this purpose, we use an established read simulator, ART [17], to generate test datasets, with properties matching realistic NGS data. We simulate reads off of the genomes of the rabies virus (12 Kbp)[18] and S. cerevisiae (12 Mbp)[19] for our behavioral analysis of Pluribus. Experiments are performed on varyling levels of coverage, ranging from $10\times$ to $30\times$.

3.5.3 Correctability Results

We perform our error classification analysis on the simulated datasets as described in Section 2. As stated previously, the error correction process is performed in a series of rounds. However, we limit the correctability analysis to the behavior during the first round of correction. We impose this limit since the corrections in earlier rounds influence the correctability of other errors in later rounds. Therefore, errors can be classified unambigiously only within the first round of correction. Fig. 6 shows the relative rates of each of the defined error classes in stacked bar plots for each genome and level of coverage simulated. Note that the range of fraction exceeds 1 on the two plots. The range from 0 to 1, shows the relative rates of the classes “Undetectable”, “Indistinguishable”, “Single Suffix Correctable”, “Multi-Suffix Correctable”, and “Uncorrected”. The summation of

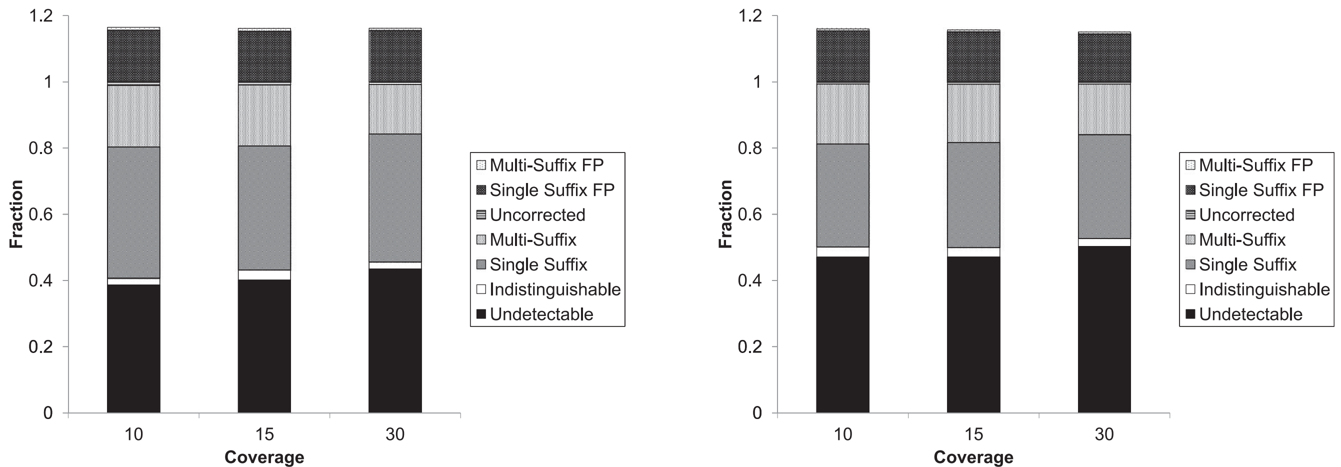

Fig. 6. "Correctability analysis" for the simulated rabies and yeast data. The left plot shows the results for the rabies virus and the right plot shows the results for yeast. Fraction exceeds 1 in order to show the relative rates of false positives to true errors.

these classes is equal to the total number of true errors in the read set and is treated as the base for comparison. The bands stacked on top of that range are "Single Suffix False Positive" and "Multi-Suffix False Positive" which represent instances where a false positive correction will be applied to the read set. We show how the false positive rate compares to the total number of errors in the reads by stacking them on top of all the true errors.

The "Single Suffix" and "Multi-Suffix" bands (third and fourth bands from the bottom respectively) in the plots of Fig. 6 show all of the errors that PLURIBUS is able to properly correct in the first round of correction on the simulated data. The "Single Suffix" band shows that around 39.6 percent of sequencing errors in the S. cerevisiae datasets and 31.1 percent of them in the rabies datasets are able to be corrected by choosing any arbitrary suggested correction in C. The "Multi-Suffix" band shows how often the voting scheme is able to identify the valid correction when C contains both valid and invalid candidate corrections, 14.9 to 18.6 percent of all errors.

As shown in the experiments with real sequencing data, PLURIBUS provides excellent precision, but it has a lower rate of recall. The bottom band of each stacked bar plot represents the fraction of errors that the algorithm is unable to find valid corrections for. We find that suffix trie based algorithms are unable to identify valid corrections for around 40 percent of the errors in the rabies datasets, and this rate increases slightly with increasing coverage. The same trend is exhibited for the yeast datasets. The increase in the rate of the undetectable errors due to coverage can be explained by the fact that for these experiments τ was held constant. PLURIBUS's low recall is a consequence of this relatively large fraction of errors for which valid corrections cannot be found. Some of these errors that are undetectable during this first round may become correctable in later rounds, but as seen in the real sequencing data experiments a large number remain uncorrected.

4 CONCLUSION

In this paper, we propose a suffix trie based method, PLURIBUS (available at `http://compbio.case.edu/pluribus`), for correcting sequencing errors in Next Generation Sequencing data. The key innovation of the proposed method is the voting scheme used to identify the best correction to apply to a read. This approach is enabled by performing correction in a read-driven manner to identify multiple manifestations of the sequencing errors of that read. We found that PLURIBUS is a conservative correction method that has higher levels of precision than other state-of-art correction tools but has lower levels of recall in comparison. We explored combining PLURIBUS and RACER together to correct data, and we found that correcting the data with PLURIBUS either before of after correcting the data with RACER improved correction performance. Overall, correcting the data with PLURIBUS before RACER worked better than the reverse order. We attribute this behavior to PLURIBUS's low rate of false positive corrections, which minimizes any compounding effects due to any false positives induced each time the data is corrected. The trends in correction performance were realized in the quality of the contigs generated in assembly. Assembling data that was corrected by PLURIBUS and then corrected by RACER had the overall best performance on the real sequencing datasets. These performance trends were consistent across most datasets and even as coverage was artificially reduced by random downsampling.

ACKNOWLEDGMENTS

The authors would like to thank Matthew Ruffalo (CWRU), Gökhan Yavaş (CWRU), Marzieh Ayati (CWRU), and Wojciech Szpankowski (Purdue) for many useful discussions, and anonymous reviewers who reviewed a preliminary version of this paper for ACM-BCB 2013 for their useful comments and suggestions. This work was supported, in whole or in part, by US National Science Foundation awards IIS-0916102 and BIO-1124962, National Institutes of Health (NIH) awards R01-AI114814 and R01-LM011247 from the National Libraries of Medicine, the Center for Science of Information (CSoI), an US National Science Foundation Science and Technology Center, under grant agreement CCF- 0939370, and by American Cancer Society Grant 123436-RSG-12-159-01-DMC.

REFERENCES

[1] J. Shendure and H. Ji, "Next-generation DNA sequencing," *Nature Biotechnology*, vol. 26, no. 10, pp. 1135–1145, 2008.

[2] M. Ruffalo, T. LaFramboise, and M. Koyutürk, "Comparative analysis of algorithms for next-generation sequencing read alignment," *Bioinf.*, vol. 27, no. 20, pp. 2790–2796, Oct. 2011. [Online]. Available: http://dx.doi.org/10.1093/bioinformatics/btr477

[3] D. R. Zerbino and E. Birney, "Velvet: Algorithms for de novo short read assembly using de Bruijn graphs," *Genome Res.*, vol. 18, no. 5, pp. 821–829, 2008.

[4] P. A. Pevzner, H. Tang, and M. S. Waterman, "An eulerian path approach to DNA fragment assembly," in *Proc. Nat. Academy Sci.*, vol. 98, no. 17, pp. 9748–9753, 2001. [Online]. Available: http://www.pnas.org/content/98/17/9748.abstract

[5] D. Kelley, M. Schatz, and S. Salzberg, "Quake: Quality-aware detection and correction of sequencing errors," *Genome Biol.*, vol. 11, no. 11, 2010, Art. no. R116. [Online]. Available: http://genomebiology.com/2010/11/11/R116

[6] J. Schröder, H. Schröder, S. J. Puglisi, R. Sinha, and B. Schmidt, "Shrec: A short-read error correction method," *Bioinf.*, vol. 25, no. 17, pp. 2157–2163, 2009. [Online]. Available: http://bioinformatics.oxfordjournals.org/content/25/17/2157.abstract

[7] L. Ilie, F. Fazayeli, and S. Ilie, "Hitec: Accurate error correction in high-throughput sequencing data." *Bioinf.*, vol. 27, no. 3, pp. 295–302, 2011. [Online]. Available: http://dblp.uni-trier.de/db/journals/bioinformatics/bioinformatics27.ht ml#IlieFI11

[8] Y. Liu, J. Schröder, and B. Schmidt, "Musket: A multistage k-mer spectrum-based error corrector for illumina sequence data," *Bioinf.*, vol. 29, no. 3, pp. 308–315, 2013.

[9] L. Ilie and M. Molnar, "Racer: Rapid and accurate correction of errors in reads," *Bioinf.*, vol. 29, no. 19, pp. 2490–2493, 2013.

[10] X. Yang, K. S. Dorman, and S. Aluru, "Reptile: Representative tiling for short read error correction," *Bioinf.*, vol. 26, no. 20, pp. 2526–2533, Oct. 2010. [Online]. Available: http://dx.doi.org/10.1093/bioinformatics/btq468

[11] L. Salmela, "Correction of sequencing errors in a mixed set of reads," *Bioinf.*, vol. 26, no. 10, pp. 1284–1290, 2010.

[12] L. Salmela and J. Schröder, "Correcting errors in short reads by multiple alignments," *Bioinf.*, vol. 27, no. 11, pp. 1455–1461, 2011.

[13] K. R. Bradnam, J. N. Fass, A. Alexandrov, P. Baranay, M. Bechner, I. Birol, S. Boisvert, J. A. Chapman, G. Chapuis, and R. Chikhi, "Assemblathon 2: Evaluating de novo methods of genome assembly in three vertebrate species," *GigaScience*, arXiv:1301.5406, vol. 2, no. 1, p. 1, 2013.

[14] M. Molnar and L. Ilie, "Correcting illumina data," *Briefings Bioinf.*, vol. 16, no. 4, pp. 588–599, 2015, Doi. 10.1093/bib/bbu029.

[15] A. Gurevich, V. Saveliev, N. Vyahhi, and G. Tesler, "Quast: Quality assessment tool for genome assemblies," *Bioinf.*, vol. 29, no. 8, pp. 1072–1075, 2013.

[16] D. M. Savel, T. LaFramboise, A. Grama, and M. Koyutürk, "Suffix-tree based error correction of NGS reads using multiple manifestations of an error," in *Proc. Int. Conf. Bioinf., Comput. Biol. Biomed. Informat.*, 2013, Art. no. 351.

[17] W. Huang, L. Li, J. R. Myers, and G. T. Marth, "Art: A next-generation sequencing read simulator," *Bioinf.*, vol. 28, no. 4, pp. 593–594, 2012. [Online]. Available: http://bioinformatics.oxfordjournals.org/content/28/4/593.abstract

[18] K.-K. Conzelmann, J. H. Cox, L. G. Schneider, and H.-J. Thiel, "Molecular cloning and complete nucleotide sequence of the attenuated rabies virus sad B19," *Virology*, vol. 175, no. 2, pp. 485–499, 1990.

[19] S. R. Engel, et al., "The reference genome sequence of saccharomyces cerevisiae: Then and now," *G3: Genes-Genomes-Genetics*, vol. 4, no. 3, pp. 389–398, 2014.

Daniel Savel received the BS degree in computer engineering from Case Western Reserve University in 2010. His research interests include sequence analysis and computational genomics.

Thomas LaFramboise received the BS degree in theoretical mathematics from the University of Michigan, Ann Arbor. In 2002, he retrained in biostatistics, earning a master's degree from the Harvard School of Public Health, which led to a postdoctoral position in cancer genomics at the Dana-Farber Cancer Institute and the Broad Institute of Harvard/MIT. He received the PhD degree in theoretical mathematics from the University of Illinois at Urbana-Champaign. Subsequently, he spent several years as an assistant and associate professor in the Mathematics Department, Kenyon and Marietta Colleges in Ohio. He joined the Department of Genetics and Genome Sciences at Case Western Reserve University in August 2006. His group is currently developing and applying methods to mine high-dimensional data sets, with the goal of generating hypotheses regarding gene function in tumor initiation, progression, and metastasis.

Ananth Grama received the BEng degree in computer science from the Indian Institute of Technology at Roorkee in 1989, the MS degree in computer engineering from Wayne State University in 1990, and the PhD degree in computer science from the University of Minnesota in 1996. He has been with Purdue University since then. He currently serves as a professor in the Computer Science Department, and director of the Computational Science and Engineering, and Computational Life Sciences graduate programs. His research interests include parallel and distributed systems, data analytics, and applications in life sciences.

Mehmet Koyutürk received the PhD degree in computer science from Purdue University in 2006. He is T. & D. Schroeder associate professor in the Department of Electrical Engineering and Computer Science, Case Western Reserve University. His research focuses on the analysis of biological networks, systems biology of complex diseases, and computational genomics. He currently serves as an associate editor for the *IEEE/ACM Transactions on Computational Biology and Bioinformatics* (*TCBB*) and the *EURASIP Journal on Bioinformatics and Systems Biology*.

▷ **For more information on this or any other computing topic, please visit our Digital Library at** www.computer.org/publications/dlib.