Western Graduate&PostdoctoralStudies

Western University
Scholarship@Western

Electronic Thesis and Dissertation Repository

December 2012

Error Correction in Next Generation DNA Sequencing Data

Michael Z. Molnar
The University of Western Ontario

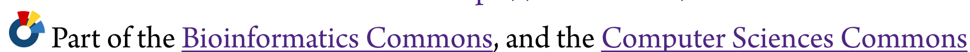
Supervisor
Dr. Lucian Ilie
The University of Western Ontario

Graduate Program in Computer Science

A thesis submitted in partial fulfillment of the requirements for the degree in Master of Science

© Michael Z. Molnar 2012

Follow this and additional works at: https://ir.lib.uwo.ca/etd



Part of the Bioinformatics Commons, and the Computer Sciences Commons

Recommended Citation
Molnar, Michael Z., "Error Correction in Next Generation DNA Sequencing Data" (2012). *Electronic Thesis and Dissertation Repository*. 991.
https://ir.lib.uwo.ca/etd/991

This Dissertation/Thesis is brought to you for free and open access by Scholarship@Western. It has been accepted for inclusion in Electronic Thesis and Dissertation Repository by an authorized administrator of Scholarship@Western. For more information, please contact tadam@uwo.ca.

ERROR CORRECTION IN NEXT GENERATION DNA SEQUENCING DATA

(Spine title: RACER)

(Thesis format: Monograph)

by

Michael Molnar

Graduate Program in Computer Science

A thesis submitted in partial fulfillment
of the requirements for the degree of
Masters of Science

The School of Graduate and Postdoctoral Studies
The University of Western Ontario
London, Ontario, Canada

© Michael Molnar 2012

THE UNIVERSITY OF WESTERN ONTARIO
School of Graduate and Postdoctoral Studies

CERTIFICATE OF EXAMINATION

Examiners:

. . . .
Dr. Sylvia Osborn

. . . .
Dr. Stuart Rankin

. . . .
Dr. Roberto Solis-Oba

Supervisor:

. . . .
Dr. Lucian Ilie

The thesis by

Michael Molnar

entitled:

Error Correction in Next Generation DNA Sequencing Data

is accepted in partial fulfillment of the
requirements for the degree of
Masters of Science

. . . .
Date

. . . .
Chair of the Thesis Examination Board

Abstract

Motivation: High throughput Next Generation Sequencing (NGS) technologies can sequence the genome of a species quickly and cheaply. Errors that are introduced by NGS technologies limit the full potential of the applications that rely on their data. Current techniques used to correct these errors are not sufficient due to issues with time, space, or accuracy. A more efficient and accurate program is needed to correct errors from NGS technologies.
Results: We have designed and implemented RACER (Rapid Accurate Correction of Errors in Reads), an error correction program that targets the Illumina genome sequencer, which is currently the dominant NGS technology. RACER combines advanced data structures with an intricate analysis of data to achieve high performance. It has been implemented in C++ and OpenMP for parallelization. We have performed extensive testing on a variety of real data sets to compare RACER with the current leading programs. RACER performs better than all the current technologies in time, space, and accuracy. RACER corrects up to twice as many errors as other parallel programs, while being one order of magnitude faster. We hope RACER will become a very useful tool for many applications that use NGS data.

Keywords: bioinformatics, DNA sequencing, next-generation sequencing, high-throughput technology, error correction, genome assembly

Contents

Certificate of Examination ii

Abstract iii

List of Figures vi

List of Tables vii

1 Introduction 1

2 Next Generation Sequencing 3
2.1 Sanger Method 3
2.2 454 Genome Sequencer 4
2.3 Applied Biosystems SOLiD 4
2.4 Illumina Genome Analyzer 6

3 Current Programs 7
3.1 EULER 7
3.1.1 Spectral Alignment Problem 7
3.2 Coral 8
3.2.1 Indexing Reads 8
3.2.2 Multiple Alignments 8
3.2.3 Correcting Reads 9
3.2.4 Complexity 9
3.2.5 Choosing Parameters 10
3.3 Quake 10
3.3.1 Counting *k*-mers 10
3.3.2 Localizing Errors and Correcting Reads 11
3.3.3 Heuristics 11
3.4 Reptile 11
3.4.1 Methods 12
3.5 HSHREC 12
3.5.1 Methods 12
3.5.2 Data Structures 12
3.5.3 Algorithm 13
3.6 HiTEC 14

3.6.1 Correcting Errors 14
3.6.2 Statistical Analysis 15
3.6.3 HiTEC Algorithm 15

4 RACER **17**
4.1 Implementation 17
4.1.1 Storing the Reads 17
4.1.2 Threshold and *k*-mer Length Calculation 20
4.1.3 Hash Table and Hashing Function 20
4.1.4 Storing *k*-mers and Counting Adjacent Bases 21
4.1.5 Correcting Reads 23
4.1.6 Heuristics 23
4.2 Testing 23
4.2.1 Data Sets 23
4.2.2 Evaluation 24
4.2.3 Results of Raw Data Sets 26
4.2.4 Results of Mapped Data Sets 26

5 Conclusion **33**

Bibliography **34**

Curriculum Vitae **35**

List of Figures

2.1 Pyrosequencing in the 454 Genome Sequencer [7]. 4
2.2 Sequencing by ligation used by SOLiD [7]. 5
2.3 Colour space encoding used by SOLiD [7]. 6
2.4 Reversible terminator imaging from Illumina [7]. 6

3.1 Multiple sequence alignment in Coral [10]. 9
3.2 Correcting a substitution error [9]. 13
3.3 Correcting indels [9]. 13
3.4 Suffix array showing supp(u,T)=5 and supp(u,A)=1 [2]. 14

4.1 An example of FASTA and FASTQ format. 18
4.2 2-bit encoding of TACGTCGA. 19
4.3 Hash function example. 21
4.4 Finding k-mers and adjacent bases. 22
4.5 From top left by rows; Escherichia coli, Pseudomonas aeruginosa, Haemophilus influenzae, Saccharomyces cerevisiae. 25
4.6 Drosophila melanogaster and Caenorhabditis elegans. 25

List of Tables

2.1 Details of NGS technologies [5][15]. 3

4.1 Data sets used for testing. 24
4.2 Accuracy in % using raw data. 27
4.3 Run time in seconds using raw data. 28
4.4 Peak space used in MB for raw data. 29
4.5 Accuracy in % using mapped data. 30
4.6 Run time in seconds using mapped data. 31
4.7 Peak space used in MB for mapped data. 32

Chapter 1

Introduction

Our ability to determine the genome of any species has revolutionized our understanding of biology. The Sanger method [11], developed by Frederick Sanger in the 1970's, was the first widely used method to determine the genome of a species. This technique has been refined and automated over the years but is still costly, time consuming, and produces a small amount of data per run. Recently, the use of Next Generation Sequencing (NGS) technologies has been replacing Sanger sequencing as the preferred method of genome sequencing [6]. One reason is that NGS technologies have a much lower cost per DNA base, as low as 40,000 times cheaper [5]. They also produce as much as 600,000 times more information per run than the Sanger method [5]. The disadvantage is that NGS technologies produce shorter pieces of a genome, called *reads*, and have more errors than the Sanger method [1].

There are two types of errors that are predominant in NGS technologies; substitution errors happen when a single base has been changed to a different base, and indels which are stretches of the genome that have been inserted or deleted from a read. Correcting these errors can greatly increase the performance of applications that use this data such as *de novo* genome sequencing, re-sequencing, and metagenomics. The companies that produce NGS technologies include 454 Life Sciences, Applied Biosystems, Helicos BioSciences, Illumina, and Ion Torrent. The most prevalent NGS technology is Illumina, due to its low cost and high throughput. The Illumina platform tends to make substitution errors [6], and for this reason most software developed for correcting errors focuses on substitution errors [6]. Some of the most successful error correction programs that correct substitution errors include Coral [10], HiTEC [2], Quake [3], Reptile [16], and SHREC [12].

Most of the other NGS technologies produce indels, which cannot be corrected by most of the current software [15]. SHREC has been updated to handle indels, and the new program is called Hybrid-SHREC (HSHREC)[9]. The only other stand alone program that can correct indels is Coral. Another important update in HSHREC is the ability to handle mixed data sets from multiple NGS technologies. Coral can correct one type or the other, but not a mixed input file.

Genome assembly software can also correct errors in reads, but most do not correct the reads directly. Most programs build a graph using overlaps in the reads, or parts of the reads, and search the graph for errors. One of the first genome assembly programs designed to use short reads from NGS technologies was EULER [1]. EULER breaks the reads into small chunks, called k-mers, and uses their $k-1$ overlaps to determine the sequence of a genome. The

data structure used to find the overlaps in the k-mers is called a *de Bruijn* graph. Another type of genome assembly software uses overlaps between entire reads to determine the sequence of the genome. This approach is referred to as the *overlap-layout-consensus* method. Both methods are sensitive to errors in the reads, and correcting the errors before running the programs can significantly improve their performance [6].

All read error correction programs require a certain level of coverage in order to make accurate corrections: if G is the length of the genome being sequenced, and N is the total number of bases that were sequenced, then the coverage is the interger value of N/G. Theoretically, if the coverage of a data set is c, then each DNA base in the genome should be represented c times in the data set, assuming the coverage is uniform. In practice, the coverage of the genome is not uniform, some regions will be represented more than others. For all programs to be able to make significant corrections the coverage needs to be above a certain threshold [9]. If the coverage is high and errors are infrequent, then the correct base is the one that appears the majority of the time [10]. The minority base at that position is considered an error and changed to the base most seen at that position [10].

RACER uses the k-mer approach to correcting substitution errors, and uses a fast and space efficient data structure to make corrections. The statistical analysis of the data provides the information needed to make highly accurate corrections. Testing has shown RACER to be the best performing software in terms of time, space, and accuracy.

A review of the sequencing technologies will be discussed first, followed by on overview of the current error correction software. A detailed explanation of the implementation of our program will follow, then the results of our testing.

Chapter 2

Next Generation Sequencing

NGS technologies are massively parallel sequencing machines that have provided an immense increase in DNA sequencing throughput. Current technologies produce high quality short reads of 25-700 base pairs (bp) in length, which is much shorter than the Sanger method which produces read lengths $\approx$ 1000bp [5]. The details of the current NGS technologies are listed in Table 2.1. The advantage of the NGS technologies is that the total number of base pairs sequenced in a single run is as many as 600,000 times more than the Sanger method. The most commonly used NGS technologies include the Genome Sequencer from 454 Life Sciences, the SOLiD system from Applied Biosystems, and the Genome Analyzer from Illumina [8].

2.1 Sanger Method

Before sequencing a genome it must be copied many times for it to been seen on the gel that separates the DNA fragments. This is accomplished by using the polymerase chain reaction (PCR), which makes many copies of the genome. After PCR the Sanger method uses inhibitors that terminate synthesized DNA chains at specific nucleic acids to determine the last letter of the DNA fragments. The procedure is run once for each letter of DNA, so that all the fragments will end in the same letter for each run. The fragments are fractionated in parallel by gel electrophoresis, and the pattern of bands that appears represents the sequence of the DNA fragment [11]. Methods were later developed to automate the process using phosphorescent chain terminators, and a capillary system to separate the fragments [14].

Table 2.1: Details of NGS technologies [5][15].

Company	Read Length (bp)	Throughput	Technique	Dominant Error Type
Illumina	36, 50, 100	105-600Gb	Reversible terminator	Substitution
Applied Biosystems	35, 60, 75	7-9Gb	Sequencing by ligation	Substitution
Helicos BioSciences	25-55	21-35Gb	Single molecule sequencing	Insertion Deletion
454 Life Sciences	700	700Mb	Sequencing by synthesis	Insertion Deletion
IonTorrent	200	1Gb	Ion semiconductor sequencing	Insertion Deletion

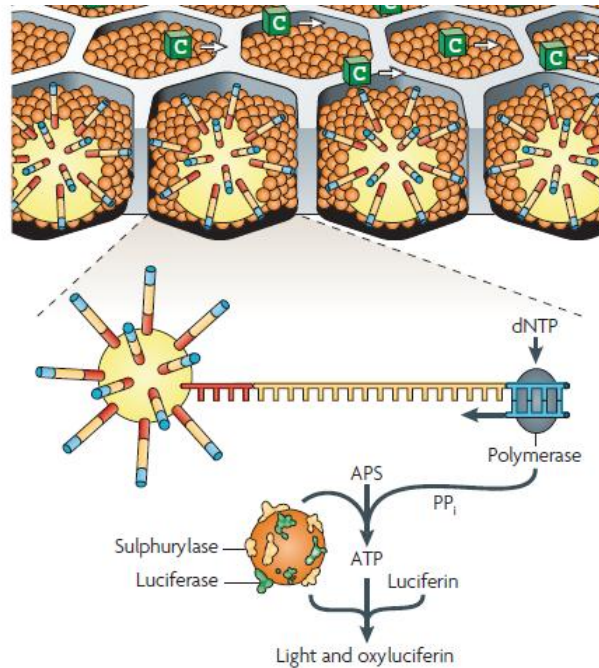


Figure 2.1: Pyrosequencing in the 454 Genome Sequencer [7].

2.2 454 Genome Sequencer

The 454 Genome Sequencer uses a technique called *pyrosequencing* to read DNA sequences. This technique relies on the firefly enzyme luciferase to emit light to detect the DNA bases that are incorporated. The fragments are amplified by a technique called emulsion PCR. Repeated rounds of emulsion PCR result in about one million copies of each DNA fragment [6].

This technique cannot interpret six or more consecutive stretches of the same nucleotide, which causes insertion and deletion errors in the reads. Since each round is nucleotide specific the amount of substitution errors is low. This method produces reads 250bp long which are screened to remove poor quality sequences. The final output results in approximately 100Mbp of data per run. An assembly algorithm called Newbler is incorporated which can assemble viral and bacterial genomes with high quality [6].

2.3 Applied Biosystems SOLiD

The SOLiD system uses an adapter ligated fragment library and emulsion PCR similar to the 454 Genome Sequencer to amplify DNA fragments. Instead of the luciferase enzyme, the SOLiD system uses DNA ligase to detect the nucleotides that are incorporated into the DNA fragment. This procedure is shown in Figure 2.2. This technique can produce fragments 25-35bp long, with approximately 2-4Gbp of data per run. Quality values are assigned to the base calls and poor quality reads are removed [6].

Most NGS technologies output the reads in base space, which consists of the letters in DNA {A, C, G, T} and possibly an ambiguous letter to indicate a base that could not be deter-

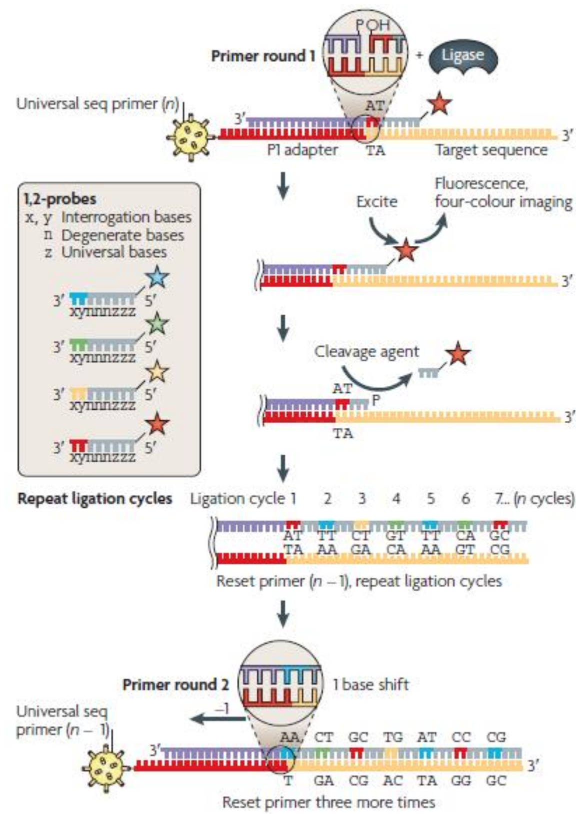


Figure 2.2: Sequencing by ligation used by SOLiD [7].

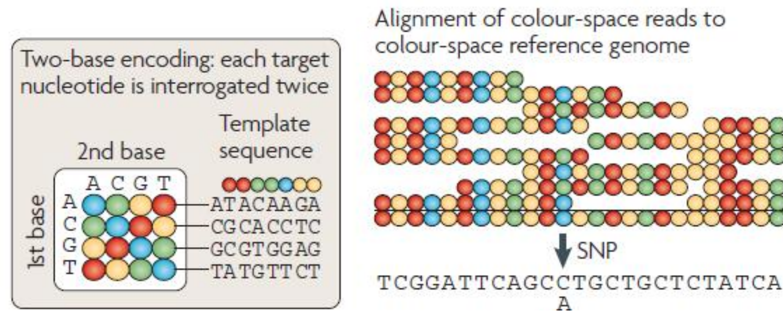


Figure 2.3: Colour space encoding used by SOLiD [7].

mined. The SOLiD genome sequencer outputs reads in colour space, which determines bases in overlapping pairs so that each base is sequenced twice [6]. An example of the colour space encoding is shown in Figure 2.3.

2.4 Illumina Genome Analyzer

The Illumina Genome Analyzer relies on an adapter library that is attached to a surface called a *cluster station*. The fragments are amplified in a process called *bridge amplification*, which uses DNA polymerase to make copies of the DNA fragments. Each cluster of fragment copies contains approximately one million copies of the original fragment. This is needed in order to produce an image that is strong enough to detect with a camera [6].

The system adds all four nucleotides simultaneously to the fragments. Each nucleotide contains a unique fluorescent label and a chemical that blocks any other base from being incorporated. An image is taken of the incorporated base before the next nucleotide is added. After each imaging step the chemical block is removed so that the next nucleotide can be added. This procedure is repeated to determine the sequence of the fragment. An example of the imaging is shown in Figure 2.4. The current model can read fragments up to approximately 36-100bp long [15].

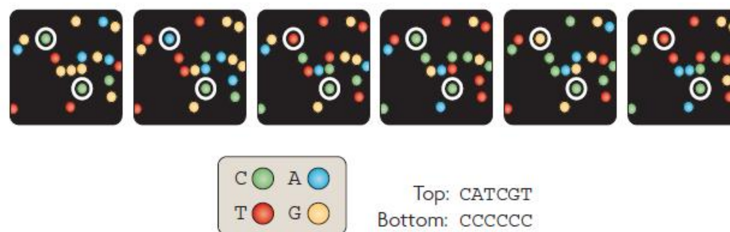


Figure 2.4: Reversible terminator imaging from Illumina [7].

Chapter 3

Current Programs

3.1 EULER

The EULER [1] program was developed to assemble genomes using short reads from NGS technologies. EULER breaks reads into smaller pieces, called k-mers, which are all of equal length, and builds a de Bruijn graph to detect errors in reads and assemble the genome. A de Bruijn graph is a directed graph of overlaps between k-mers with the length of the overlap equal to k-1. In the graph each k-mer is represented by a node and an edge is an overlap with another node. Errors complicate a de Bruijn graph, which increases running time and space, so EULER tries to correct the reads before building the de Bruijn graph.

3.1.1 Spectral Alignment Problem

If we let G_k be the set of k-mers in a genome G, then a read has errors if it contains k-mers that are not in G. EULER tries to change all reads so that every k-mer in a read is found in G. This transformation is referred to as the spectral alignment problem (SAP).

In the SAP there is a collection of k-mers, T, and a string. A string is called a T-string if all the k-mers in the string are in T. The goal of the SAP is to find the minimal number of corrections to change all strings to T-strings. Error correction is done by setting T to G_k and finding the spectral alignment of each read.

If the genome of a sample is known, then determining which k-mers are in the genome is a trivial task. In *de novo* genome sequencing the sampled genome is not known, so EULER uses the number of times a k-mer appears to determine if it is in G_k. A threshold M is approximated and any k-mer that appears more than M times is referred to as *solid*, and less than M times is called *weak*. A read that contains all solid k-mers is referred to as a *solid read*, and the goal of EULER is to correct the reads so that they are all solid reads.

Low coverage data sets require that M be set low so that valid k-mers are not considered weak, but this will cause correct reads to be considered incorrect. Increasing the length of k will help reduce the rate of false positives, but makes it more difficult to determine which changes are correct to transform a read to a solid read.

EULER uses a dynamic programming solution that chooses a small M while eliminating many false positives by considering multiple changes to find the optimal solution. Since the

genome is not known, the set of all k-mers in the genome is approximated.

The ends of reads tend to have more errors than the rest of the read, and it is difficult to determine the correct changes to the ends of reads, so reads with erroneous ends are trimmed. The set of all k-mers is updated after each iteration of corrections, and reads are deleted if they are not able to be corrected after the last iteration. The assembly of the genome begins after the error correction.

3.2 Coral

Coral [10] is able to correct both substitution errors and indels. This is accomplished by using multiple sequence alignments (MSA) between short reads to detect errors. The parameters for scoring the alignments can have a significant impact on the quality of the alignments, and therefore the quality of the corrections. The parameters that can be set are gap penalty and mismatch penalty. Coral also provides default parameter selection based on the platform that is used. For substitution errors the gap penalty needs to be set very high to prevent indels in the alignments. For indels Coral suggests to set the gap and mismatch penalties to be equal.

3.2.1 Indexing Reads

Coral starts by indexing all the k-mers in each read and their reverse complements. A hash table is used to store the k-mers and the reads that contain each k-mer. It is redundant to save both the k-mer and its reverse complement, so to save space only the lexicographically smaller of the two is used to store the information. Coral also does not store any k-mers that contain ambiguous letters.

3.2.2 Multiple Alignments

After indexing the k-mers, the next step is to build the MSA between reads. Each alignment starts with one read which is called the *base read*. All the reads that share at least one k-mer with the base read are found using the k-mer hash table. This set of reads, along with the base read, is called the *neighbourhood* of the base read.

Computing the MSA of large data sets can be very time consuming. Coral uses some heuristics to speed up the alignments. If the neighbourhood of the base read is very large or very small then Coral does not perform any alignments. If the neighbourhood is very large then the reads likely came from different regions in the genome and the MSA will take a long time to compute because the alignments will be very poor. If the neighbourhood is small then there is likely not enough coverage to make any significant corrections.

Another heuristic used saves time by not correcting reads that have been corrected before. A read can be in several neighbourhoods, so if a read has already been corrected in a previous alignment, then Coral tries to align it to the consensus without errors in the region of the k-mer. If it aligns perfectly then the rest of the read is not aligned to save time.

A few other heuristics are used to save time. If a read shares at least one k-mer only once with the base read then a banded Needleman-Wunsch alignment is performed. If a read has many errors compared to the consensus then it stops aligning the read and moves on to the next

Column	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19	20	21	22	23	24	25	26	27	28	29	30	31	32	33	34	35	36	37	38	39	40	41	42	43	44	45	46
Consensus	G	T	C	A	G	A	A	-	G	T	G	A	G	C	G	T	G	G	C	A	T	T	A	A	C	C	C	T	T	G	A	T	A	C	C	A	C	C	G	G	T	T	C	A	A	C
Read 1	G	T	C	A	G	A	A	-	G	T	G	A	G	C	G	T	G	G	C	A	T	T	A	A	C	C	C	T	T	G	A	T	A													
	40	40	40	40	40	40	40	**40**	40	40	40	40	40	40	40	40	40	40	40	40	40	40	40	40	40	40	40	40	40	40	40	40	40													
Read 2			C	A	G	A	A	A	G	T	G	A	G	C	G	T	G	G	C	A	T	T	A	A	C	C	C	T	T	G	A	T	A	C	C											
			33	36	40	40	40	34	40	40	40	40	40	40	40	40	40	40	40	40	40	40	40	40	40	40	40	40	40	40	40	40	40	40	40											
Read 3			C	A	G	A	A	-	G	T	G	A	G	C	G	T	G	G	C	A	T	T	A	A	C	C	C	T	T	G	A	T	A	C	C											
			40	40	40	40	40	**40**	40	40	40	40	40	40	40	40	40	40	40	40	40	40	40	40	40	40	40	40	40	40	40	40	40	40	40											
Read 4									G	T	G	A	G	C	G	T	G	G	C	A	T	T	A	A	C	C	-	T	T	G	A	T	A	C	C	A	C	C	G	G						
									40	40	40	40	40	40	18	40	40	40	40	40	40	33	40	30	40	22	**31**	40	40	40	40	40	40	40	40	40	40	40	40	40						
Read 5																T	G	G	C	A	T	T	A	A	C	C	C	T	T	G	A	T	A	C	C	A	C	C	G	G	T	T	C	A	A	
																40	40	40	40	40	40	40	40	40	40	40	40	40	40	40	40	40	37	40	40	40	40	40	40	40	19	40	40	40	40	
Read 6																	G	G	C	A	T	T	A	C	C	C	C	T	T	G	A	T	A	C	C	A	C	C	G	G	T	T	C	A	A	C
																	22	40	40	17	40	40	40	33	40	40	40	40	40	40	40	40	40	40	40	40	40	40	40	27	40	40	40	40	40	40

Figure 3.1: Multiple sequence alignment in Coral [10].

one. Finally, if gaps are not allowed then a gap-less alignment is performed between a read and the consensus sequence.

3.2.3 Correcting Reads

To correct the reads Coral calculates the number of misaligned positions for each read compared to the consensus sequence. If the quality of the alignment is above 1 - e, where e is the expected error rate, then it will be used to correct the aligned reads. Figure 3.1 shows an example of a MSA. For each position in the consensus sequence, the support threshold for each position is calculated by the number of times each letter occurs, divided by the total number of reads aligned at that position. If a read differs from the consensus at any position, then the letter is changed in the read provided the support is above the threshold.

If quality scores are available then an extra check is made before correcting. If there is a gap in an aligned read, then the quality score of the gap is set to the average of the quality scores of the bases flanking the gap. A weighted support is then calculated by dividing the sum of the quality scores of the bases that agree with the consensus, with the sum of all the quality scores at that position. A correction is made if a base is above the quality threshold. The quality score of a corrected base is set to the average of the quality scores of the bases that agree with the consensus at that position.

3.2.4 Complexity

The MSA is the dominant part of the time complexity for Coral. Assume the combined total length of the reads is M, the maximum number of reads in a neighbourhood is L, and the longest read length is r. The worst case runtime is $O(MrL)$ if gaps are allowed, and $O(ML)$ if only mismatches are allowed. Since the total number of bases is M, then there cannot be more than $O(M)$ k-mers. Therefore, the space for the hash table that stores the k-mers is bounded by $O(M)$. The space complexity for computing the MSA is $O(Lr + r^2)$. The overall space complexity is $O(M)$.

3.2.5 Choosing Parameters

The choice of k has a significant impact on the quality of the corrections. The value of k should be set so that each k-mer appears in the genome only once. The authors suggest that $k \geq [\log_4 G]$, where G is the length of the genome. The value of the error rate e should be set to the expected error rate of the input data. If the value of e is set too high then Coral could over correct reads because even a poor MSA will be corrected.

The support threshold can also be set, and should be related to the coverage of the data set. If the coverage is c then the support threshold should be set to $(c - 1) / c$. If the threshold value is set too low Coral will over correct, and if it is set too high it will under correct. The quality value threshold should be set in a similar way. Coral has predefined settings for Illumina data sets to correct substitution errors, and for 454 data sets to correct indels. Coral is not able to correct mixed data sets that contain both substitution errors and indels.

3.3 Quake

Quake [3] relies on k-mer coverage and quality scores to correct reads. Quake uses a method of evaluating the quality of a k-mer based on the associated quality scores for each base. The reasoning behind this approach is that coverage alone does not take into account high coverage regions with errors, and low coverage regions without errors. This can have an effect on the choice of cutoff values for correcting k-mers. Weighing the k-mers based on quality scores helps since low coverage true k-mers will still have high quality scores, and high coverage k-mers with errors will have low quality scores. Even k-mers that appear once with high quality scores will be trusted with this approach, whereas normal k-mer counting would consider this to be an erroneous k-mer.

3.3.1 Counting k-mers

Quake begins by counting all the k-mers in the data set. The user must select the value of k, which has a significant impact on the performance of Quake. An equation is provided to determine an appropriate choice for k. If G is the size of the genome then the authors suggest setting the value for k such that:

$$\frac{2G}{4^k} \approx 0.01 \tag{3.1}$$

Which simplifies to:

$$k \approx log_4 200G \tag{3.2}$$

Using this equation most bacterial genomes will have a k-mer length around 14 or 15, and the human genome has a k-mer length of 19. Quake uses quality score information and approximates the distribution of k-mers based on their quality values. Quake then determines a cutoff value which separates trusted k-mers from untrusted k-mers.

3.3.2 Localizing Errors and Correcting Reads

Once the cutoff has been set, all reads with untrusted *k*-mers are considered for correction. In most cases the errors are localized in a small region of a read. To find an error in a read, an intersection of the untrusted *k*-mers is used to localize the search. This works for reads with a few errors, but not for reads with many errors, or low coverage regions that are below the cutoff. There is also a problem when the errors are near the end of a read. If this is the case, Quake considers every base covered by the right most trusted *k*-mer, and left most trusted *k*-mers to be correct. These localizing heuristics are used to reduce the run time need to find errors in the reads.

Once the region of a read that needs to be corrected is found, Quake tries to find the corrections that make all *k*-mers in that region trusted. To define the likelihood of a set of corrections let O represent the observed nucleotides of the read, and A represent the actual nucleotides of the sequenced fragment. Quake tries to evaluate the conditional probability of a potential assignment of A, given the observed nucleotides O.

Quake models errors to allow for biases in base substitutions that are known to exist for Illumina data sets. The initial pass of the reads only makes unambiguous corrections, and leaves low quality reads for the next pass. After the first pass, modifications are made to reduce the variance of the error model to correct the remaining ambiguous errors.

3.3.3 Heuristics

Quake uses some heuristics to reduce space and running time. Reads from repeat regions may have multiple sets of valid corrections, with a small difference in the likelihood of each correction, so a true correction is ambiguous. To address this issue Quake continues past the threshold to ensure that another valid set does not exist. Another likelihood threshold of 0.01 is used in this situation.

A large majority of computation time is spent on the lowest quality reads, which have many potential sets of corrections to consider. Quake saves time correcting low quality reads by prescreening the erroneous region, and stops correcting if the region is filled with low quality scores. Reads with a region containing ≥ 13 positions with a probability of error $> 1\%$ are not corrected. For reads with regions containing ≥ 9 such positions, the likelihood ratio threshold is increased to 10^{-3}.

3.4 Reptile

Reptile [16] uses the *k*-spectrum approach similar to EULER, but multiple *k*-mer decompositions are used along with information from neighbouring *k*-mers to make corrections. Reptile also uses quality score information if it is available. Reptile is designed to correct short reads with substitution errors.

3.4.1 Methods

Reptile tries to create approximate multiple alignments which allows for substitutions. This could be done by considering all reads with pairwise Hamming distance less than a set threshold, but such alignments are difficult. Reptile finds the multiple alignments by only aligning k-mers in the reads. The size of k is chosen so that the expected number of occurrences of any k-mer in the genome should be no more than one.

Reptile uses contextual information to help resolve errors without increasing k. It is expected that if a read contains errors then there should be a tiling of reads that cover the true read. The tiling is done by taking the reads that share a k-mer and aligning them from the location of the k-mer. The locations in the tiling that have higher coverage than those of lower coverage are used to make corrections.

The main disadvantage of Reptile is that the user is required to set the parameters. This requires running scripts and analyzing the results to determine the proper parameters. The input must also be changed to the format that is accepted by Reptile. Most other programs set the parameters automatically, aside from the size of the k-mer, and most accept the standard formats of read files.

3.5 HSHREC

HSHREC is an updated version of SHREC [12]. SHREC was designed to correct substitution errors, and HSHREC is able to correct both substitution errors and indels. It is also able to correct data sets that have a mix of both types of errors.

3.5.1 Methods

HSHREC [9] assumes that the input contains k reads randomly sampled from a genome with read length n, where n can vary in length. The errors can be substitutions, or indels. It is assumed the errors are randomly distributed and that the coverage is sufficient for correction.

Errors are detected in a read by aligning it to the other reads using a generalized suffix *trie*, which is a graph that contains all the suffixes of the reads. It is assumed that most reads that overlap an erroneous read will not have the same error. The erroneous region of the read has a low weight in the trie and this is the area that will be corrected.

Single nucleotide polymorphisms (SNPs) are single nucleotide changes that differ between individuals of the same species. SNPs can appear as errors in the multiple alignment since they create columns in the alignment where reads do not match. Since errors at the SNP location will only be in a few reads, and SNPs will be present in several reads, it is possible to differentiate them.

3.5.2 Data Structures

Let R be the set of reads from the input, and their reverse complements. A unique number from 1 to $2k$ is concatenated to the end of each string in R so that each read has a unique suffix. The edges of the trie are labeled with DNA letters, and an ambiguous letter N, {A, C, G, T, N}, and

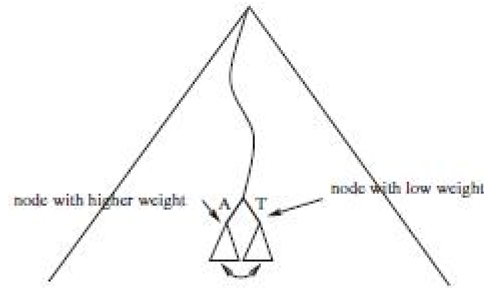


Figure 3.2: Correcting a substitution error [9].

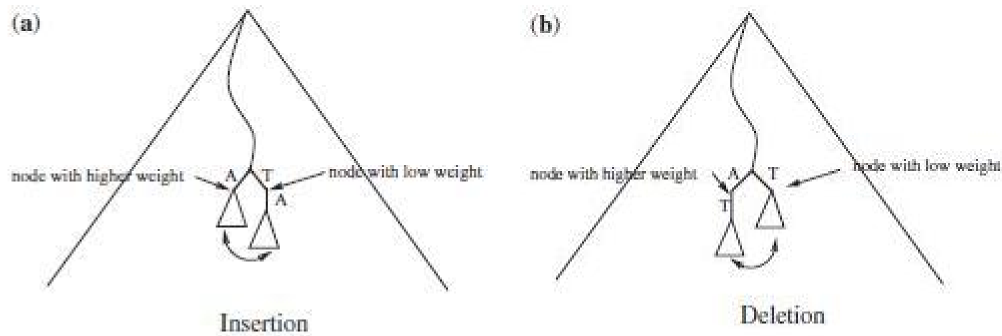


Figure 3.3: Correcting indels [9].

each node may only have one child labeled with the same character. The concatenation of edge labels from the root to a node is called a *path-label*. For any suffix of a sting in *R*, there is a *path-label* that contains that suffix. The *weight* of a node in the trie is the number of leaves in the subtrie that is rooted at that node, and is the number of suffixes in that subtrie. The *level* of a node is the length of the path from the root to the node.

In the top levels of the trie almost all nodes have four or five children, and further down the trie almost all nodes have only one child. If a child at this level has more than one child then it is likely an error. The node with the lower weight is likely the erroneous base. Below this level the weight can be close between each child, and it is too difficult to distinguish between the correct base and the erroneous base. The HSHREC algorithm traverses the trie to identify errors at the intermediate levels of the trie. It first tries to correct each error with a substitution, and the remaining errors are treated as insertions or deletions.

3.5.3 Algorithm

The original SHREC algorithm starts by building the generalized suffix trie and tries to correct substitution errors starting at an intermediate level in the trie. To correct a node that is determined to have an error, SHREC compares the subtrie rooted at the low weight node to the subtries rooted at the siblings of the node. This is shown in Figure 3.2. If a correction cannot be made then the read is marked as erroneous.

Indels also cause extra branching in the generalized suffix trie. An insertion creates a low

weight node and deleting the node causes the children rooted at the node and their siblings to be merged. A comparison of the subtries before and after the deletion determine if the deletion was a proper way to correct the node. If it is then the reads that contain that suffix are changed to reflect the changes in the subtrie. Deletions are handled in the same manner, as long as there is another node at the same level with a higher weight. These two situations are shown in Figure 3.3.

3.6 HiTEC

HiTEC (High Throughput Error Correction) [2] uses a similar type of data structure as SHREC, but instead uses a suffix array that is built using a string of the reads and their reverse complements. This is a more time and space efficient data structure than the suffix trie used in SHREC. HiTEC uses less space than SHREC, and the serial HiTEC is faster than SHREC in parallel. HiTEC has been shown to be the most accurate algorithm for correcting substitution errors. This is accomplished by automatically setting its parameters using statistical analysis, and varying the length of the sections it corrects in each read.

The suffix array can be computed in linear time and space. However, HiTEC uses a suboptimal algorithm that works better in practice than the optimal solution. HiTEC uses an array that stores the length of the longest common prefix (LCP) between consecutive suffixes in the suffix array. Since HiTEC only requires bounded LCP values, the LCP array is computed directly.

3.6.1 Correcting Errors

To correct errors HiTEC assumes that for a genome G with length L, each read is a random string over Σ with length l. The reads $r_1, r_2, ..., r_n$ are produced from G with a per-base error rate of p. Any read with a letter not in Σ is discarded. The reads and their reverse complements are stored in a string $R = r_1\$\bar{r}_1\$r_2\$\bar{r}_2\$...r_n\$\bar{r}_n\$$, where \$ is a letter not in Σ.

The correction algorithm for HiTEC assumes that a read r_i, starting at position j of the genome, contains an error in position k and that the previous w positions, $r_i[k-w..k-1]$, are correct. When there is an error in a read at a location in the genome, it is assumed that most

Figure 3.4: Suffix array showing supp(u,T)=5 and supp(u,A)=1 [2].

of the other reads will have sampled that location in the genome correctly. If there is enough of a majority of one base at that position, then the erroneous base is changed to the base that appears most at that position. For $a \in \Sigma$, the support of u for a is $\text{supp}(u, a)$, which is the total number of occurrences of the string ua in R. An example of this is shown in Figure 3.4.

3.6.2 Statistical Analysis

It is easy for HiTEC to calculate the support values, given the suffix array, since all occurrences of u supporting the same letter are consecutive in the suffix array. The clusters correspond to witnesses of a given length w, and can be found easily using the LCP array.

Repeats in the genome can cause problems with a small u. Since u could be present in many places, there is a higher chance of an error not being detected. A long u will be less likely to appear in the genome, but will be covered by fewer reads, thus reducing its support. HiTEC first estimates the support given by a witness u. A witness is considered correct if it occurs above a threshold, and erroneous otherwise.

Statistical analysis helps compute a threshold, T, that is used to detect errors. There is often an interval of values where every T is good. This interval grows when the error rate decreases. To cover the case when low coverage causes very low values for T, the value of T is increased by a constant of two.

Some reads will have no w consecutive correct positions, which makes it impossible to fit a correct witness at any position. Therefore, such reads cannot be corrected, so the number of such reads is approximated and then the algorithm is adjusted to correct most of those reads. The number of uncorrectable reads decreases as the length of w decreases, but the probability of seeing the witness increases, which causes correct positions to be changed incorrectly. The number of incorrect changes, or destructible reads, is approximated by HiTEC.

Since lowering the witness length w decreases the number of uncorrectable reads $U(w)$, but increases the number of destructible reads $D(w)$, a value for w must be found that minimizes $U(w)+D(w)$. Theoretically, this provides the highest accuracy, but in practice a combination of values that are close to the optimal witness length causes almost no correct reads to be changed. HiTEC uses a variation of witness lengths based on the optimal witness length to achieve high accuracy. The details of how these values are computed is discuss in the HiTEC paper.

3.6.3 HiTEC Algorithm

The user must supply the length of the genome and the error rate of the data set. An approximation of the length of the genome is probably known by the user, and an approximate value of the error rate should be provided by the sequencing machine.

For each witness u, if there is no ambiguity in the correct letter then the correction is made. If there is ambiguity, then the next two letters are checked in order to decide how to correct the read. After a certain number of iterations of corrections there are very few bases that are corrected. This will take extra time but adds little accuracy, so HiTEC stops correcting when the number of bases changed during one iteration is less than 0.01% of the total number of bases, or after nine iterations or corrections.

Since HiTEC only requires modest coverage to make corrections, datasets with high coverage are split into several sets with lower coverage that are independently corrected. This saves

space which allows HiTEC to correct large data sets with high coverage. The disadvantages of HiTEC are that it does not correct reads with ambiguous letters, it can only correct data sets if the reads are all the same length, and it does not run in parallel mode.

Chapter 4

RACER

The most accurate read error correction program to date is HiTEC. Unfortunately, the time and space used for the suffix array in HiTEC are not the most efficient. The goal of this thesis is to correct reads with the same level of accuracy as HiTEC or higher, but reduce the time and space needed to make the corrections. A pilot implementation to accomplish this task, called HiTEC2, was previously completed by Lankesh Shivanna [13]. This thesis presents the fully functioning tool, called RACER (Rapid and Accurate Correction of Errors in Reads).

4.1 Implementation

RACER uses the same approach as HiTEC to correct reads, but the implementation is different, dramatically improving the running time and space, but also increasing the accuracy. RACER replaces the suffix array approached used in HiTEC with a more time and space efficient hash table. The hash table stores the k-mers in each read, and the total times each base appears before and after each k-mer. The optimal k-mer length is calculated with a similar statistical analysis as HiTEC. After the k-mers and the counters have been calculated, the reads are corrected based on the counts. The way RACER has been implemented allows the reads to be corrected repeatedly without having to recalculate the k-mers and counters after each iteration, as long as the same k-mer length is used.

To save space, RACER encodes the input sequences using two bits to represent each base. Storing the bases as characters would require eight bits per bases, so this approach is four times more space efficient at storing the reads. The k-mer and its reverse complement must be searched, and only the one with the smaller encoding key is stored. This decreases the amount of space needed in the hash table by half. Finally, RACER is able to correct data sets that have varying read lengths, and runs in both serial and parallel mode.

4.1.1 Storing the Reads

The reads from NGS technologies contain the four letters of the DNA alphabet, and bases that could not be determined are usually represented by the letter N. Any character in the input that is not from the DNA alphabet is randomly replaced by a letter from the DNA alphabet. RACER assumes the input files are either in *FASTA* or *FASTQ* format. An example of both

FASTA

```
>SRR123456.789 length=36
TAAATCCTCGTACAACCCAGATGGCAACCCATTACC
```

FASTQ

```
@ SRR123456.789 length=36
TAAATCCTCGTACAACCCAGATGGCAACCCATTACC
+ SRR123456.789 length=36
IIIIIIIIIIIII3IIII$I-IIBCIEIE8*??=)1
```

Figure 4.1: An example of FASTA and FASTQ format.

types of input is shown in Figure 4.1. If the reads are in FASTA format then there are two lines for each read. The first line of each read always begins with the ”>” symbol, followed by a string that identifies the read. The second line contains the DNA letters of the read, which can vary in length. If the reads are in FASTQ format then each read will occupy four lines. The first two lines for each read are similar to FASTA files, except that the first character is the ”@” symbol instead of the ”>” symbol. The first character in the third line of each read is either a ”+” or ”-” symbol to indicate if the read is in either the $5' \rightarrow 3'$ direction, or the $3' \rightarrow 5'$ direction. The fourth line for each read is the quality score of each base. The quality score is a quantitative way of determining how likely the base at that position is correct. Each sequencing platform has a unique numbering system for their quality scores.

Performing bit operations on data is much faster than using operations on characters. The logical bit operations used in RACER are AND, OR, and NOT. The AND operation is used in RACER with a mask to extract *k*-mers from the reads. A mask is a sequence of bits used to filter out unwanted bits from another sequence of bits. A mask contains zeros in positions that are not wanted, since the results will always be zero regardless of what is in the other sequence of bits. A mask is set to one in positions that are wanted, since the results will be whatever was in that position in the sequence of bits.

The 2-bit encoding of the DNA bases in RACER represents A as 00, C as 01, G as 10, and T as 11. Any ambiguous base is randomly converted to one of the four DNA bases before encoding it. Some other programs replace ambiguous bases with a single DNA letter, but we found that this can create problems when correcting. If there are many long stretches of ambiguous bases in the reads, then replacing them by all A’s will cause a *k*-mer with all A’s to be above the set threshold. Therefore, the stretch of A’s will not be corrected even when they should, causing problems for programs that use the corrected data such as *de novo* genome assemblers. This is avoided by randomly replacing the ambiguous letters.

The bases are encoded in such a way that finding the reverse complement of a base or *k*-mer can be found quickly by using the NOT logical operation. A NOT operation flips all the bits so that 0’s become 1’s, and 1’s become 0’s. The DNA letter A is the complement of T, so performing a NOT on 11 will results in 00, and from 00 to 11. The same is true for G and C, which flips the bits from 10 to 01, and from 01 to 10. This implementation is more time efficient than a character array, since bit operations are faster than comparing characters

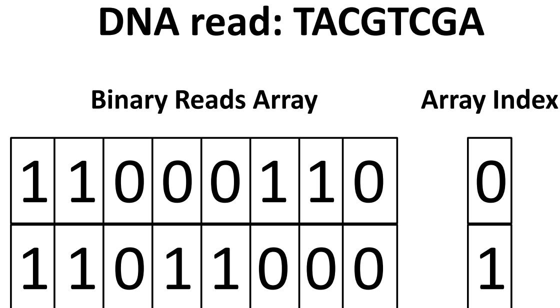


Figure 4.2: 2-bit encoding of TACGTCGA.

to determine the reverse complement.

The reads are stored sequentially in an unsigned 8-bit integer array, which is initialized to zero for each element. This means that each element of the array can store up to four DNA bases from a read. Most machines are byte addressable, which means the smallest number of bits they can access is eight bits, so storing each 2-bit base requires a mask and bit shifting.

Consider a read that contains the DNA letters TACGTCGA. Each element in the array that will contain the encoded read initially contains all 0's. The first letter is a T and is encoded to 11, then shifted left 6 positions and logically OR'd with the array element. The OR operation will put a 1 in the result if either of the operands contains a 1, and 0 otherwise. This array element will now contain 1100 0000. The next base is an A and is encoded to 00, then shifted left 4 positions and logically OR'd with the array element. This array element will now contain 1100 0000. The next base is a C and is encoded to 01, then shifted left 2 positions and logically OR'd with the array element. This array element will now contain 1100 0100. The next base is a G and is encoded to 10, then logically OR'd with the array element. This array element will now contain 1100 0110. The result of storing TACGTCGA in the binary reads array is shown in Figure 4.2.

At this point the 8-bit integer value holds 4 bases and is now full, so the index value of the array is incremented. The process is repeated until all the bases in the read are encoded. If the read length is not a multiple of 8 then the last element of the array will not be filled. The rest of the bits of that element will be left as 0's. At most there will be 6 unused bits for each read, which is still a much more space efficient way to store reads compared to using a character array. An integer array is used to store the length of each read so that the number of bits not used in the last array element for each read can be calculated.

Many data sets contain reads that are mostly ambiguous bases. These reads do not contain much reliable information from the genome sampled, so correcting them would waste time. RACER does not correct reads if more than half the bases in the read are ambiguous bases. The integer array that is used to store the read length is a positive value if the read has less than half ambiguous bases, and a negative length if the read has more than half ambiguous bases. The read is not removed from the final output, leaving the decision to use it or not to the next application.

4.1.2 Threshold and k-mer Length Calculation

In order to correct the reads with high accuracy, RACER requires three parameters to be calculated; a threshold for determining and correcting an error, a k-mer length K that will minimize the number of false positives, and a k-mer length k that will maximize the number of corrections. RACER uses statistical analysis to determine the best possible values of the parameters similar to HiTEC. A first improvement is that all statistical computations of RACER are performed by the program itself, eliminating the previous need for a C++ statistical library being installed.

One of the main reasons that HiTEC is able to produce such high accuracy is that it varies the witness length used to correct the reads. RACER does the same with the k-mer length to achieve a high level of accuracy. The combination of k-mer lengths was chosen experimentally based on the accuracy results from the testing data sets. RACER uses a maximum of eight iterations of corrections with the following k-mer lengths:

$$k + 1, k + 1, K + 1, K + 1, k - 1, k - 1, K + 1, K + 1 \tag{4.1}$$

The first step to correcting the reads is to build the hash table with the k-mers and the counters for the letters before and after each k-mer. The next step is to correct the reads with the information from the hash table. The way RACER has been implemented, the hash table can be used again without rebuilding it, if the k-mer length stays the same. RACER uses the same k-mer length for two iterations to save time building the hash table.

4.1.3 Hash Table and Hashing Function

Different strategies are used to store and retrieve information for all applications. It is best if any piece of stored information can be accessed in constant time. It is also important to use the smallest possible space to store the information. An array could be used to store the k-mers, but once the array is full it would have to be rebuilt, and if the array was sorted then inserting the new element would cause the index where some elements are stored to change. For a small data set with no sorting this is not a significant problem, but in our applications the data sets are very large and need some way to access the data quickly.

Large data sets require arrays that are large enough to store the initial data, and any data that may be added afterwards. The number of elements is usually not known in advance, so an estimate must be made for the initial array size. The initial array size should have more elements than the initial data size in order to add elements afterwards. There must also be a fast way to find elements in the array. A type of array that can accomplish this is called a *hash table*, and the function that is used to store and retrieve elements in a hash table quickly is called a *hash function*.

A hash table has two main parts, the key and the value the key points to. The key is found using the hash function, and the key is the index of the array where the value is stored. Hash tables with good hash functions can store and retrieve elements in constant time. There is a problem when we try to add an element with the same key as another element already stored in the hash table. This problem is called a *collision*, and it is a common issue with hash tables.

One way to deal with collisions is by using open addressing. It requires a search of the hash table for an empty entry to store the value being inserted. The three most common open

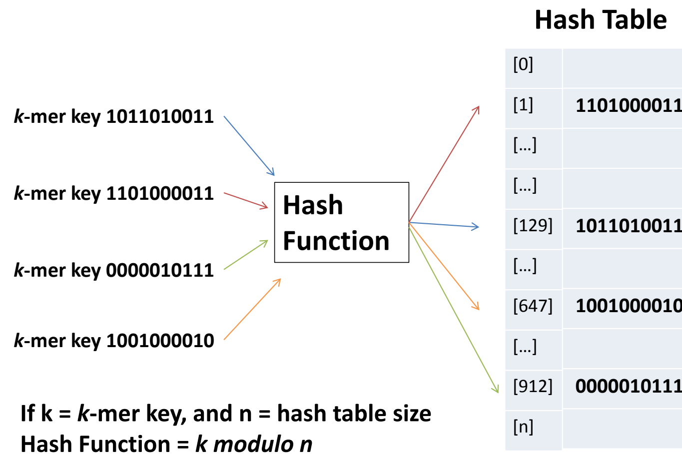


Figure 4.3: Hash function example.

addressing solutions to collisions are *linear probing*, *quadratic probing*, and *double probing*. Good probing techniques should try to distribute values evenly throughout the hash table to avoid collisions.

Linear probing is the simplest way to find an empty location to store a new element, and it is the type of probing used in RACER. If there is a collision at index i then the next index at $i+1$ is searched, if that element is occupied then the next element at $i+2$ is searched. This continues sequentially until an empty location is found to insert the new element. This approach is fast because of cache effects, but tends to cluster elements.

Quadratic probing is similar to linear probing, except that instead of searching the next element in the hash table, a quadratic polynomial determines the next index searched. This method is slower than linear probing, since it does not take advantage of cache effects. Although, there is less clustering than the linear method which can improve performance for bad hash functions. Double probing requires a second hash function to handle collisions. If there is a collision using the main hash function, then the second hash function is used repeatedly until an empty location is found.

The 2-bit representation of each k-mer is used as a unique binary number that represents the input key for the hash function. The hash function in RACER takes the input key modulo the hash table size, and the result is the index in the hash table to store the k-mer. RACER uses a hash table size that is a prime number, and initial collisions are reduced since the greatest common factor is 1. The initial size of the hash table in RACER is set to the first prime number that is greater than nine times the genome size.

4.1.4 Storing k-mers and Counting Adjacent Bases

RACER uses a hash table with linear probing to store k-mers and the counters for the adjacent bases. The k-mers are stored in a 64-bit unsigned integer array, which limits the length of a k-mer to a maximum length of 32. The counters are stored in an 8-bit unsigned integer array.

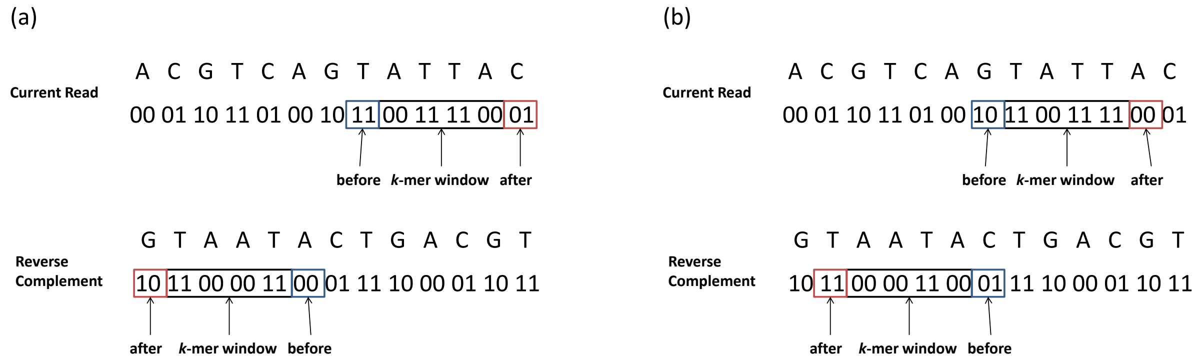


Figure 4.4: Finding *k*-mers and adjacent bases.

Each block of eight elements in the counters array stores the number of times each of the four DNA bases appears before and after each *k*-mer. The first four elements represent the four possible bases before the *k*-mer, and the next four elements represent the four possible bases after the *k*-mer. Since the counters array is 8-bits, the maximum *k*-mer occurrences that can be counted is 255. If there is a letter that appears more than 255 times it is assumed to be correct and the counter stays at 255. Once the hash table is half full it is increased to the size of the prime number closest to twice the previous hash table size. The smaller hash table is deleted before creating the larger hash table to save space, and the *k*-mers and counters are recalculated.

The process of finding *k*-mers and incrementing the counters starts by copying the current read into a temporary array. The size of the array is set to twice the size of the longest read, since two bits are needed for each letter in the read. The last 64 bits of the current read are loaded into the current 64-bit window. Another 64-bit window is used to store the reverse complement of the current 64-bit window. Once the end of the 64-bit window is reached, the next 64 bits of the read and its reverse complement are loaded into the temporary arrays.

A *k*-mer window is created which is twice the *k*-mer length, since each base is encoded using two bits. This *k*-mer window is then aligned with the rightmost end of the 64-bit window. The *k*-mer is obtained with a logical AND operation between the *k*-mer mask and the 64-bit window. The *k*-mer mask contains bit values of 1 inside the *k*-mer window and 0 elsewhere. A similar procedure is used for the reverse complement of the current 64-bit window. The bases that are before and after the *k*-mer are extracted using a similar masking procedure. Each mask is set to all 0's, except for the two bit locations where the before or after base is located. An example of this procedure is shown in Figure 4.4. After storing a *k*-mer and incrementing its counters, Figure 4.4(a) , the *k*-mer windows shifts two bits to find the next *k*-mer in the read, Figure 4.4(b).

Storing the information for both the forward and reverse complements is redundant, so space is halved by only storing the smaller of the two integer values. The hash function is then used to store or find the *k*-mer in the hash table. If the *k*-mer already exists then the appropriate counters are incremented, if it is not in the hash table it is added.

The next *k*-mer is found by shifting the *k*-mer window two bits to the left in the current 64-bit window, and two bits to the right in the reverse complement. The *k*-mer is stored if it

is not in the hash table and the counters are incremented, then the window is shifted two bits again. This continues until all the k-mers in the read are found. If l is the read length and k is the k-mer length, then the number of k-mers in a read is $l - k + 1$.

4.1.5 Correcting Reads

After finding the k-mers and incrementing the counters, RACER begins correcting the reads. Correction starts by finding the k-mers in a read with the same technique described previously. The counters for each k-mer are used to determine if a correction needs to be made in the base before or after the k-mer, and to decide what is the correct base. If the count for the before or after base is less than the threshold it is considered an error, and the counters are searched for a base that is above the threshold. If there is another base above the threshold then the erroneous base is replaced with the correct base. If there is more than one base that is above the threshold then no corrections are made. If the total count for the before or after base is above the threshold it is considered correct and no corrections are made.

The hash table and the corrected read are updated before the next k-mer in a read is considered for correcting. An advantage of this implementation is that once an erroneous base is corrected, the next k-mer that is considered will contain the corrected base. This allows RACER to correct more than one error in a read in one iteration, and eventually exceed the accuracy of HiTEC.

4.1.6 Heuristics

RACER has a minimum amount of four iterations of corrections, and a maximum amount of eight iterations of corrections. After four iterations, if the number of corrections is below 0.01% of the total number of reads, then RACER stops correcting. This is because there are not likely any more significant changes that can be made. Extensive testing was performed to evaluate the performance of RACER compared to the current leading technologies.

4.2 Testing

4.2.1 Data Sets

To test the performance of RACER we obtained fifteen data sets from the Sequence Read Archive (http://trace.ncbi.nlm.nih.gov/Traces/sra/). The information for each data set is listed in Table 4.1. The goal was to find good data sets that vary in read length, coverage, and genome size. There was also a large variation in the error rates, most notably the high error rate of E.coli 3.

Many of the publications from the competing software include correction of mapped data sets. Mapping a data set requires the reference genome of the species from which the reads were obtained. Each read is aligned to the reference genome with a certain number of mismatches allowed per read. The reads that were able to align are kept, and the reads that did not align are removed. This procedure filters out reads with many errors, which improves the performance of the error correction software. This is not done in practice, but for completeness

Table 4.1: Data sets used for testing.

Data Sets	Read Length	Error Rate	Genome Length	Number of Reads	Coverage
L.lactis	36	0.52%	2,598,144	4,370,050	60.55
T.pallidum	35	0.89%	1,139,417	7,133,663	19.13
E.coli 1	75	0.65%	4,639,675	3,454,048	55.83
B.subtilis	75	0.58%	4,215,606	3,519,504	62.62
E.coli 2	75	0.62%	4,639,675	4,341,061	70.17
P.aeruginosa	36	0.09%	6,264,404	9,306,557	53.48
E.coli 3	47	3.65%	4,771,872	14,408,630	141.92
L.interrogans 1	100	0.26%	4,338,762	7,066,162	162.86
L.interrogans 2	100	0.21%	4,277,185	7,127,250	166.63
E.coli 4	36	0.46%	4,771,872	20,816,448	157.04
H.influenzae	42	0.39%	1,830,138	23,935,272	549.29
S.aureus	76	1.75%	2,901,156	25,551,716	669.36
S.cerevisiae	76	0.72%	12,416,363	52,061,664	318.67
C.elegans	100	0.35%	102,291,899	67,617,092	66.10
D.melanogaster	45,75,95	1.12%	120,220,296	101,548,652	57.43

we corrected both the raw and mapped data sets. We used a program called Burrows-Wheeler Aligner [4] to map the reads to their reference genomes using the default parameters.

Figure 4.5 shows images of the organisms corresponding to some of the smaller genomes used in our testing. The top left image is of Escherichia coli, which is one of the most studied organisms to date, due to its extensive use in Biotechnology. The top right image is of Pseudomonas aeruginosa, which is a commonly found bacteria that can cause death in humans if it infects major organs. The bottom left image is of Haemophilus influenzae, which is more commonly known as the flu. Finally, the bottom right image is of Saccharomyces cerevisiae, which is a type of yeast.

Figure 4.6 shows the images of the organisms corresponding to the two large genomes used in our testing. The left image is Drosophila melanogaster, more commonly known as the fruit fly. Due to its high reproductive rate and ease of maintenance, the fruit fly is one of the mostly widely used organisms in genetic research. The right image is of Caenorhabditis elegans, which is another highly studied organism for genetic research.

These organisms were chosen because they are all commonly studied organisms, which means the reference genome for each should be reliable. This assures that accuracy results are valid, and that there are plenty of read files available from the Sequence Read Archive.

4.2.2 Evaluation

RACER was compared to the current top performing read error correction software. This included Coral, HiTEC, Quake, Reptile, and SHREC. All programs were tested on the raw and mapped data sets. The competing programs were run according to the specifications in their respective manuals, websites, and readme files. All tests were run on the Shared Hierarchical

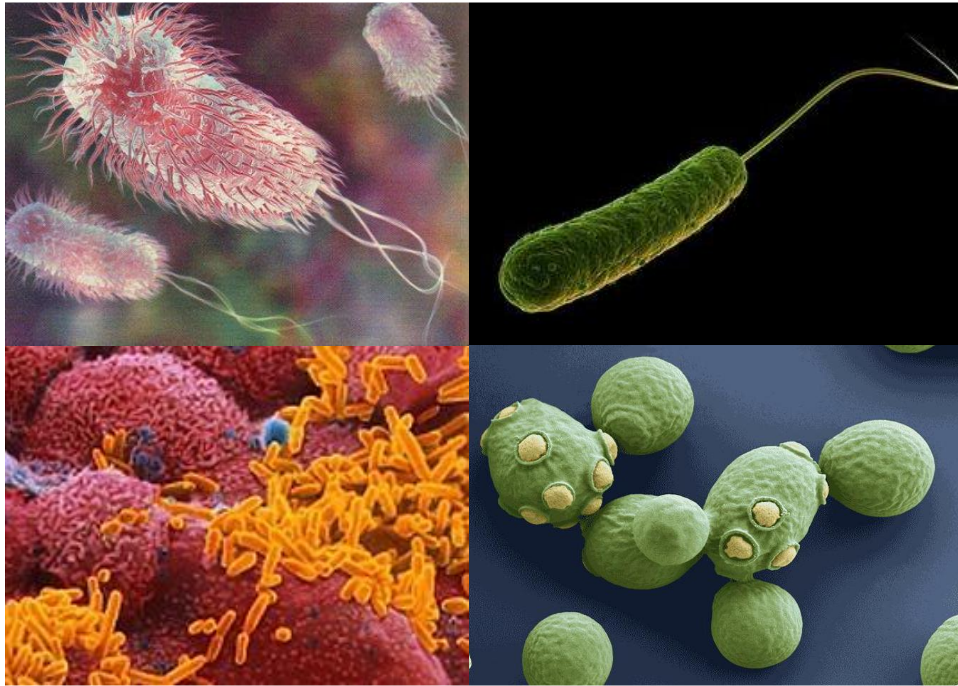


Figure 4.5: From top left by rows; Escherichia coli, Pseudomonas aeruginosa, Haemophilus influenzae, Saccharomyces cerevisiae.

Figure 4.6: Drosophila melanogaster and Caenorhabditis elegans.

Academic Research Computing Network (SHARCNET), with a HP 24 core 2.1 GHz AMD Opteron with 98GB RAM running Linux Red Hat, CentOS 5.6.

The data sets are measured in time, space, and accuracy. The time is measured in seconds, and space is the peak space reported by SHARCNET. The accuracy is calculated using the number of reads corrected (TP - true positives), the number of correct reads made incorrect (FP - false positives), and the number of reads with errors that were not corrected (FN - false negatives). The formula used to calculate the accuracy is:

$$\frac{TP - FP}{TP + FN} \tag{4.2}$$

4.2.3 Results of Raw Data Sets

The previous software with the highest accuracy was HiTEC. The results of our testing shows that RACER outperforms HiTEC in accuracy in most cases. Even for the data sets that HiTEC corrects more reads, it is by less than 1%. RACER had much better accuracy results for the data sets S.aureus and S.cerevisiae because HiTEC stopped after one iteration of corrections, whereas RACER used eight iterations of corrections. The main advantage of RACER over HiTEC is the faster running time and decreased peak space used.

RACER was the fastest program in both serial and parallel modes. Quake was the second fastest in serial mode, but RACER was twice as fast in serial mode. RACER was one order of magnitude faster than all programs in parallel mode. Coral had the largest improvement from serial to parallel mode at 15 times faster, but also required an increase in space of 12 times. RACER was not far behind in the increase in time at 11 times, but the increase in space was minimal. The overall space used was the lowest for RACER. Quake is the next best program at space reduction, but still uses more than 50% more space than RACER in parallel mode.

HiTEC was not able to correct D.melanogaster due to the varying read sizes. Quake was not able to correct H.influenzae and S.aureus due to a failed cut off value. SHREC was not able to correct L.lactis and E.coli 3 because of an error while reading the input. The rest of the missing results were due to the programs running out of space. All tests were allocated 98GB of RAM.

4.2.4 Results of Mapped Data Sets

The results for the mapped data sets is similar to the raw data sets. The difference is that the programs run faster, use less space, and have much better accuracy results. This is due to the fact that the mapped data sets have less reads, with a minimal amount of errors per read. Reptile was the only program that had a lower accuracy with the mapped data sets compared to the raw data sets. The parameters were adjusted to try and get better results, but nothing we tried improved the accuracy. This shows why it is important for a program to set its parameters automatically. Some programs require the user to input the k-mer length or the genome length, but Reptile is the only program that requires the user to set all parameters manually.

Table 4.2: Accuracy in % using raw data.

	Serial						Parallel			
Genome	Coral	HiTEC	Quake	Reptile	SHREC	RACER	Coral	Quake	SHREC	RACER
L.lactis	65.54	80.61	71.65	60.27	-	80.49	65.52	71.31	-	80.49
T.pallidum	38.55	84.45	59.46	2.65	61.79	85.75	38.54	59.24	61.50	85.75
E.coli 1	26.04	82.72	1.50	21.82	72.61	83.58	26.04	2.11	71.67	83.58
B.subtilis	59.76	80.59	53.59	64.25	41.19	82.12	59.77	53.63	40.54	82.12
E.coli 2	9.80	76.38	2.51	54.55	38.58	76.32	9.80	2.18	37.78	76.32
P.aeruginosa	79.78	78.68	7.08	68.44	63.40	85.32	79.77	30.48	63.37	85.32
E.coli 3	0.00	19.35	8.53	0.00	-	56.50	0.00	8.46	-	56.50
L.interrogans 1	48.25	60.23	49.75	35.55	55.99	59.87	48.25	49.75	55.15	59.87
L.interrogans 2	44.16	54.26	44.97	38.46	48.09	53.91	44.16	44.95	47.39	53.91
E.coli 4	58.02	85.89	81.38	0.06	77.49	86.32	58.02	81.43	77.03	86.32
H.influenzae	28.39	73.33	60.52	10.64	53.45	78.35	28.39	-	53.19	78.35
S.aureus	0.02	0.03	-	0.03	-	25.96	0.02	-	-	25.96
S.cerevisiae	2.85	0.23	6.81	11.38	9.17	12.25	2.85	6.90	8.96	12.25
C.elegans	-	-	38.88	0.21	-	56.54	-	38.87	-	56.54
D.melanogaster	-	-	35.36	0.56	-	42.95	-	35.47	-	42.95

Table 4.3: Run time in seconds using raw data.

	Serial						Parallel			
Genome	Coral	HiTEC	Quake	Reptile	SHREC	RACER	Coral	Quake	SHREC	RACER
L.lactis	1,741	852	1,694	623	-	174	208	559	-	23
T.pallidum	7,325	2,636	4,309	2,266	5,373	780	534	1,028	548	71
E.coli 1	6,330	3,074	842	2,248	6,672	1,366	512	658	787	99
B.subtilis	6,192	3,114	1,309	1,988	6,322	1,067	513	698	767	111
E.coli 2	8,427	4,001	1,109	3,306	8,504	1,322	682	727	1,348	118
P.aeruginosa	3,663	916	1,348	4,744	5,652	409	424	1,000	658	63
E.coli 3	3,960	7,862	22,874	4,509	-	2,790	738	3,596	-	483
L.interrogans 1	64,343	7,837	1,678	3,334	18,115	2,268	3,456	851	2,460	180
L.interrogans 2	69,289	8,831	1,840	3,288	18,724	2,029	3,648	1,174	2,070	163
E.coli 4	19,133	9,610	10,123	4,295	13,681	1,515	1,497	2,228	1,608	202
H.influenzae	84,724	13,562	6,108	10,309	18,736	2,127	4,282	-	2,125	266
S.aureus	142,233	3,753	-	29,496	-	8,518	8,714	-	-	1,077
S.cerevisiae	359,097	8,081	8,915	29,174	100,425	13,732	18,894	4,812	20,212	1,294
C.elegans	-	-	19,975	104,010	-	34,165	-	6,906	-	2,618
D.melanogaster	-	-	69,747	128,981	-	46,476	-	24,352	-	6,242

Table 4.4: Peak space used in MB for raw data.

	Serial						Parallel			
Genome	Coral	HiTEC	Quake	Reptile	SHREC	RACER	Coral	Quake	SHREC	RACER
L.lactis	1,996	2,979	493	579	-	515	52,544	1,619	-	722
T.pallidum	3,271	4,738	1,297	768	35,178	569	53,819	1,838	99,492	798
E.coli 1	3,810	4,895	1,230	1,765	35,987	1,437	54,355	1,944	99,394	1,773
B.subtilis	4,243	4,981	1,332	1,595	36,444	1,438	54,789	1,945	99,600	1,773
E.coli 2	4,910	6,064	1,732	3,598	36,454	1,477	55,427	2,180	99,606	1,803
P.aeruginosa	3,579	6,340	1,851	790	35,865	1,167	54,189	2,575	99,525	1,429
E.coli 3	11,934	12,755	3,364	1,914	-	6,433	62,453	4,045	-	6,944
L.interrogans 1	8,210	7,253	2,673	2,431	38,030	966	58,727	3,226	99,628	1,383
L.interrogans 2	7,982	7,343	2,800	2,322	37,961	964	58,499	3,139	99,591	1,252
E.coli 4	8,235	14,178	4,090	1,070	37,872	1,411	58,816	5,008	99,515	1,949
H.influenzae	10,231	19,132	4,253	1,060	38,142	1,278	60,811	-	99,637	1,669
S.aureus	43,981	36,117	-	8,837	-	4,561	94,499	-	-	5,109
S.cerevisiae	41,278	77,893	15,056	4,421	69,125	5,628	91,858	15,581	100,002	6,267
C.elegans	-	-	32,001	10,406	-	17,803	-	32,688	-	18,263
D.melanogaster	-	-	36,374	21,069	-	41,206	-	36,868	-	42,229

Table 4.5: Accuracy in % using mapped data.

	Serial						Parallel			
Genome	Coral	HiTEC	Quake	Reptile	SHREC	RACER	Coral	Quake	SHREC	RACER
L.lactis	75.22	92.16	81.67	0.11	84.87	92.02	75.21	81.82	84.74	92.02
T.pallidum	50.81	91.72	68.77	0.77	70.72	92.35	50.82	69.18	70.55	92.35
E.coli 1	26.49	83.08	1.51	0.07	73.05	83.91	26.49	1.65	72.14	83.91
B.subtilis	73.93	92.64	63.90	32.25	47.93	93.55	73.93	63.92	47.46	93.55
E.coli 2	11.65	78.22	1.42	0.07	40.35	77.80	11.65	1.16	39.56	77.80
P.aeruginosa	84.28	82.92	60.54	3.26	66.74	89.51	84.27	60.56	66.71	89.51
E.coli 3	3.78	77.00	45.79	0.02	56.44	82.67	3.78	45.97	55.53	82.67
L.interrogans 1	75.18	91.58	76.07	0.12	85.44	90.81	75.20	76.11	84.22	90.81
L.interrogans 2	75.35	90.05	75.33	1.61	80.31	89.27	75.34	75.47	79.44	89.27
E.coli 4	67.74	90.74	90.73	0.07	86.65	90.80	67.75	90.85	86.19	90.80
H.influenzae	48.65	80.04	69.60	8.32	60.66	84.33	48.57	-	60.45	84.33
S.aureus	0.25	0.43	32.75	0.07	40.49	47.00	0.25	32.83	39.35	47.00
S.cerevisiae	5.97	0.30	8.92	0.30	11.94	14.68	5.97	9.00	11.75	14.68
C.elegans	27.53	-	47.90	0.26	-	65.96	-	47.92	-	65.96
D.melanogaster	40.12	-	47.01	0.00	-	57.04	-	47.19	-	57.04

Table 4.6: Run time in seconds using mapped data.

	Serial						Parallel			
Genome	Coral	HiTEC	Quake	Reptile	SHREC	RACER	Coral	Quake	SHREC	RACER
L.lactis	1,667	823	1,261	1,760	2,626	157	183	551	317	20
T.pallidum	7,366	1,669	2,170	2,565	3,964	462	518	906	448	49
E.coli 1	6,017	3,069	709	2,370	6,442	951	485	640	831	92
B.subtilis	6,239	2,926	1,101	1,829	5,272	881	480	657	624	80
E.coli 2	8,534	4,028	865	2,666	9,033	1,187	702	787	1,656	186
P.aeruginosa	3,573	867	1,136	4,189	5,726	498	386	835	642	43
E.coli 3	1,705	2,375	9,053	3,205	5,232	819	252	1,747	1,041	166
L.interrogans 1	61,714	7,084	1,538	4,381	16,883	1,665	3,210	932	1,983	155
L.interrogans 2	65,612	8,062	1,777	3,006	16,459	1,318	3,419	848	2,650	155
E.coli 4	18,691	5,108	4,693	6,008	12,872	942	1,392	1,588	1,443	101
H.influenzae	26,876	7,404	4,573	8,993	17,428	1,269	4,419	-	1,876	160
S.aureus	9,214	3,928	5,567	9,528	31,471	4,901	6,388	2,857	8,101	81
S.cerevisiae	357,529	7,074	7,799	34,476	84,381	11,899	19,434	3,924	14,040	840
C.elegans	262,581	-	15,831	104,010	-	33,072	-	5,773	-	1,932
D.melanogaster	152,651	-	46,248	148,164	-	30,117	-	10,827	-	2,883

Table 4.7: Peak space used in MB for mapped data.

	Serial						Parallel			
Genome	Coral	HiTEC	Quake	Reptile	SHREC	RACER	Coral	Quake	SHREC	RACER
L.lactis	1,962	2,914	233	543	34,238	467	52,573	1,553	99,514	730
T.pallidum	3,067	4,557	449	689	34,829	478	53,616	1,616	99,538	768
E.coli 1	3,786	4,927	371	1,724	35,907	1,393	54,330	1,533	99,615	1,730
B.subtilis	3,511	4,694	424	1,680	34,837	1,389	54,056	1,713	99,648	1,727
E.coli 2	4,747	5,974	399	1,718	38,430	1,420	55,268	1,734	99,548	1,747
P.aeruginosa	3,571	6,330	311	781	35,418	1,079	54,118	1,507	99,528	1,277
E.coli 3	3,892	4,209	1,184	1,043	36,399	1,530	54,409	1,782	99,598	1,852
L.interrogans 1	7,375	6,683	611	2,268	36,964	859	57,891	2,413	99,578	1,217
L.interrogans 2	7,279	6,759	472	2,712	36,675	859	57,798	2,409	99,647	1,217
E.coli 4	7,772	13,937	819	878	36,996	1,141	58,354	2,882	99,718	1,683
H.influenzae	9,628	18,757	629	1,016	36,930	843	60,146	-	99,592	1,351
S.aureus	18,267	22,941	3,083	3,086	53,599	2,163	68,782	3,591	99,948	2,535
S.cerevisiae	34,491	65,590	3,041	3,411	50,815	3,233	85,069	10,828	99,789	3,937
C.elegans	75,096	-	6,685	10,406	-	16,765	-	24,747	-	17,271
D.melanogaster	83,163	-	21,516	17,802	-	39,482	-	24,339	-	40,272

Chapter 5

Conclusion

We have presented a new program, RACER, which is designed for correcting substitution errors in short reads from NGS technologies. The purpose of this thesis was to implement a program that was at least as accurate as HiTEC, but more time and space efficient. The current implementation of RACER has been shown to be the fastest, most accurate, and space efficient program to date. RACER scales very well with an increasing number of processors, which is important for the huge amounts of data produced by NGS technologies. Accuracy is improved with longer reads due to the statistical approach used to determine k-mer lengths based on the read length. Future improvements to RACER will include the ability to correct indels, read mixed data sets, and use MPI parallelization.

Bibliography

[1] M. Chaisson, P. Pevzner, and H. Tang. Fragment assembly with short reads. *Bioinformatics*, 20(13):2067–2074, 2004.

[2] L. Ilie, F. Fazayeli, and S. Ilie. HiTEC: accurate error correction in high-throughput sequencing data. *Bioinformatics*, 27:295–302, 2011.

[3] D.R. Kelley, M.C. Schatz, and S.L. Salzberg. Quake: quality-aware detection and correction of sequencing error. *Genome Biology*, 11:R116, 2010.

[4] H. Li and Durbin R. Fast and accurate short read alignment with Burrows-Wheeler Transform. *Bioinformatics*, 25:1754–1760, 2009.

[5] L. Liu and et al. Comparison of next-generation sequencing systems. *Journal of Biomedicine and Biotechnology*, 2012:1–11, 2012.

[6] E.R. Mardis. Next-generation DNA sequencing methods. *Annu. Rev. Genomics Hum. Genet.*, 9:387–402, 2008.

[7] M.L. Metzker. Sequencing technologies - the next generation. *Nature Genetics*, 11:31–46, 2010.

[8] G. Narzisi and B. Mishra. Comparing de novo genome assembly: The long and short of it. *PLoS ONE*, 6(4):e19175, 2011.

[9] L. Salmela. Correction of sequencing errors in a mixed set of reads. *Bioinformatics*, 26(10):1284–1290, 2010.

[10] Salmela, L. and Schröder, J. Correcting errors in short reads by multiple alignments. *Bioinformatics*, 27(11):1455–1461, 2011.

[11] F. Sanger and et al. DNA sequencing with chain-terminating inhibitors. *Proc. Natl. Acad. Sci.*, 74:5463–5467, 1977.

[12] Schröder, J. and Schröder, H. and Puglisi, S.J., et al. SHREC: a short-read error correction method. *Bioinformatics*, 25:2157–2163, 2009.

[13] Lankesh Shivanna. A fast implementation for correcting errors in high throughput sequencing data. Master's thesis, University of Western Ontario, 2011.

[14] Sanders J.Z. Kaiser R.J. et al Smith, L.M. Fluorescence detection in automated DNA sequence analysis. *Nature*, 321(6071):674–679, 1986.

[15] X. Yang, S.P. Chockalingam, and Aluru. S. A survey of error-correction methods for next-generation sequencing. *Briefings in Bioinformatics*, 2012.

[16] X. Yang, K.S. Dorman, and S. Aluru. Reptile: representative tiling for short read error correction. *Bioinformatics*, 26:2526–2533, 2010.

Curriculum Vitae

Name:	Michael Molnar
Post-Secondary Education and Degrees:	University of Western Ontario London, ON 2011-2012 M.Sc. (pending defense)
	University of Western Ontario London, ON 2001-2011 Honors Specialization in Bioinformatics (Biochemistry Concentration)
	Fanshawe College London, ON 1997-1999 Business Information Systems
Honours and Awards:	Dean's Honor List 2001
Related Work Experience:	Teaching Assistant The University of Western Ontario 20011-2012

Publications:

L. Ilie, M. Molnar. (2012) RACER: Rapid and Accurate Correction of Errors in Reads. Bioinformatics. Submitted.