Efficiently Improving the Reference Genome for DNA Read Alignment

Jacob Pritt

***Abstract*—In this project I implemented the Four-stage algorithm for updating a Burrows Wheeler transform by Salson et al. [1] as well as the Iterative Referencing algorithm developed by Ghanayim and Geiger [2]. After analyzing the accuracy and time and space requirements of these algorithms, I propose two variations with the goal of improving the space requirements of the iterative referencing algorithm. I compare the accuracy and space of these new variations to the original iterative referencing for simulated data, and conclude that it is possible to reduce the space to no longer be linear in the number of sequenced reads without compromising accuracy. Finally, I make suggestions for future research to further test and improve these algorithms.**

I. Introduction

THE fastest and most accurate way of assembling reads is to align them against a reference genome. However, often the genome from which the reads originate differs significantly from the reference genome. In sections of the genome with a high density of mutations, it can be difficult or impossible to align reads. However, if we can actively learn about common mutations from mapped reads and use this information to update our reference genome, this should allow us to more accurately align reads in mutation-dense regions. In order to implement this, I first researched and implemented the method by Salson et al. for updating an FM-index and the iterative referencing algorithm by Ghanayim and Geiger for iteratively improving the reference genome. After combining these methods to yield a more accurate aligner, we tested the accuracy and space requirements of the new method. I found that the space required by iterative referencing increases linearly with the depth of coverage, which is inefficient given that sequencing experiments have increasingly higher coverage as sequencers improve. I researched modifications of the iterative referencing algorithm that would reduce space requirements while still yielding high accuracy, and found that by saving only a small number of read mutations, normalized over the entire genome, I could reduce the space requirement to no longer depend on coverage depth, without sacrificing any noticeable accuracy.

II. Previous Work

The FM-index, and its main component, the Burrows Wheeler transform, is a very useful data structure that allows us to efficiently preprocess and compress a large reference text to quickly search for substrings [3]. Recently, FM-indexes have been effectively used to map sequenced DNA or RNA reads to a reference genome [4]. The FM-index takes $O(n^2)$ time in the length of the genome to construct, since it requires us to sort all the suffixes of the genome. This is okay though, since as a preprocessing step the index is only constructed once, allowing read mapping to then be very fast. However, in many cases the genome of origin for a set of reads differs substantially from the reference genome. In such cases, it would be beneficial to update the FM-index to more accurately reflect the genome of origin, in order to more accurately map later reads. For this purpose, Salson et al. developed fast algorithms for updating both the suffix array [5] and the Burrows Wheeler transform [1], which allows us to efficiently update the FM-index while actively mapping reads.

Ghanayim and Geiger [2] have presented an iterative referencing algorithm that efficiently updates the FM-index while mapping reads to it. This algorithm works by repeatedly mapping all possible reads to the reference genome, looking for a consensus among strands of mutated bases from the reference genome, and updating the FM-index to reflect these mutations. These steps are repeated until no new reads are mapped to the genome. Iterative referencing shows a significant improvement in accuracy over simple read matching, and demonstrates one way to effectively update the reference genome to more accuractely match the reads.

III. Methods and Software

A. FM-Index and Update

I began by implementing the FM-index (which was partially written in the sample code provided in Professor Langmead's slides). I next set about implementing the different updates (insertion, deletion, and substitution) to the index. This was by far the most challenging and time-consuming part of the project. The details of the algorithms are not clearly described in the two papers by Salson et al., and I had to figure out myself the correct way to update the suffix array and b-value checkpoints. For simplicity, I ultimately decided to store the entire suffix array since I could not figure out how to update the index while using checkpointed values.

B. Iterative Referencing

I next implemented a simplified version of the iterative referencing algorithm developed by Ghanayim and Geiger. My algorithm initially tries to match all reads to the reference genome, with up to 1 nucleotide error. All successfully-mapped reads are then compared to the reference genome and any differences are stored. Any mutations that appear in $\geq k$ mapped reads, for some threshold k, are incorporated into the FM-index to obtain an updated genome. This process is repeated on the remaining unmapped reads until less than 10% of unmapped reads are successfully mapped in 1 iteration.

C. Variations on Iterative Referencing

In order to reduce the space requirements of the iterative referencing algorithm, I developed several variations of the algorithm. The limiting space constraint is the storage of mutations found in matched reads, which grows linearly with the number of reads and the mutation frequency (see Results section for more details). I reduced the size of this list by storing mutations only among the first 10% of mapped reads. This change reduces the size of the mutations list by approximately 90%, while having a small negative impact on accuracy.

One downside of the previous approach is that it assumes a consistent coverage over the entire genome, while in reality coverage varies greatly across the genome. As a result, sections of the genome with higher coverage will have more accurate error detection, while some errors in low-coverage regions might be omitted from the list. I accounted for this problem by storing a coverage vector indicating which bases in the genome have already been covered by a mapped read. Only the first k reads to map to a base, for some small threshold k, can contribute to the list of mutations. This preserves a consistent weight of detected mutations across the entire genome while minimizing the number of stored mutations – for example, if k is $\frac{1}{10}$ of the average coverage, at most $\frac{1}{10}$ of the total mutations will be stored. Unfortunately, introducing a new coverage vector for the genome greatly increases the space requirements of the algorithm. However, we can reduce the size of this vector by splitting the genome into chunks and calculating the coverage for each chunk, thus preserving consistent coverage levels across the genome. I experimented with different sizes of chunks to see the effect on accuracy and space requirements.

IV. Results

A. Updating the FM-index

There are three ways to update the FM-index; an insertion, a deletion, and a substitution. The running times for all three of these methods are linear in the length of the genome, but in practice are much faster than this. Figure 1 shows that updating the FM-index is significantly faster than rebuilding it. This improvement in speed allows us to efficiently update the index while aligning reads.

I ran all tests on a simulated reference genome of 5000 random nucleotides. I then mutated each base in the genome with some probability, with equal likelihood of introducing an insertion, deletion, or mutation. Next, 500 reads of length 50 were drawn from random positions in the mutated genome and aligned against the reference genome. Throughout this paper, the approximate matching step supported only up to 1 error in matching. Supporting up to 2 errors per read makes it significantly harder to count common mutations in the reference genome – for example, an insertion at position 5 followed by a deletion at position 10 is the same as a deletion at position 9 followed by an insertion at position 5.

Figure 2 shows the proportion of accurately aligned reads for different mutation rates of the reference genome and read lengths. As we would expect, accuracy decreases for higher mutation rate, as more errors are introduced in each read. Similary, longer reads will contain more errors than shorter ones, so accuracy decreases as read length increases.

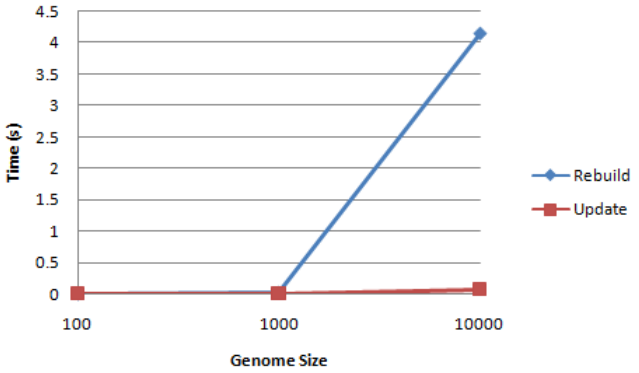

Figure 1. Time required to rebuild the FM-index vs updating it.

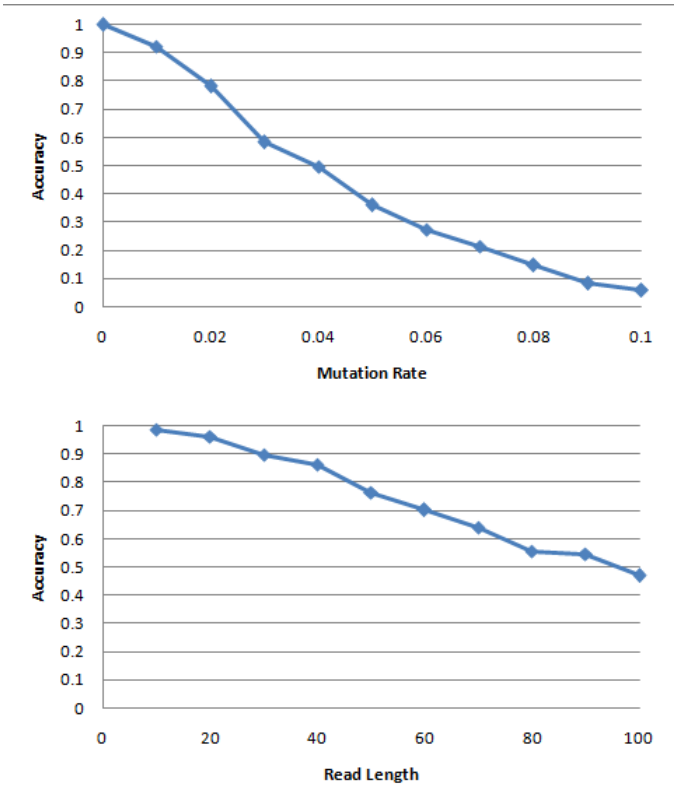

Figure 2. Accuracy of the basic alignment algorithm for different mutation rates in the genome and different read lengths.

B. Iterative Referencing

The iterative referencing algorithm [2] works as follows:

1) All possible reads are matched to the reference genome.
2) Among all matched reads, any mutations from the reference genome are counted.
3) Any mutations that appear more frequently than some given threshold are incorporated into the fm-index for the reference genome.

4) The process is repeated with any reads that were unmatched in the last round, until no new reads are mapped.

Ghanayim and Gaiger claim that this algorithm almost always finishes in less than 5 iterations, and in practice I found that it almost always concluded in 3 iterations or fewer.

I implemented the basic iterative referencing algorithm, omitting some of the features for simplicity. For example, Ghanayim and Geiger incorporate the PHRED quality score into their algorithm to improve accuracy. Since I was working with simulated data without quality scores, I omitted this step. Figure 3 compares the accuracy obtained by simple read matching to that obtained by iterative referencing.

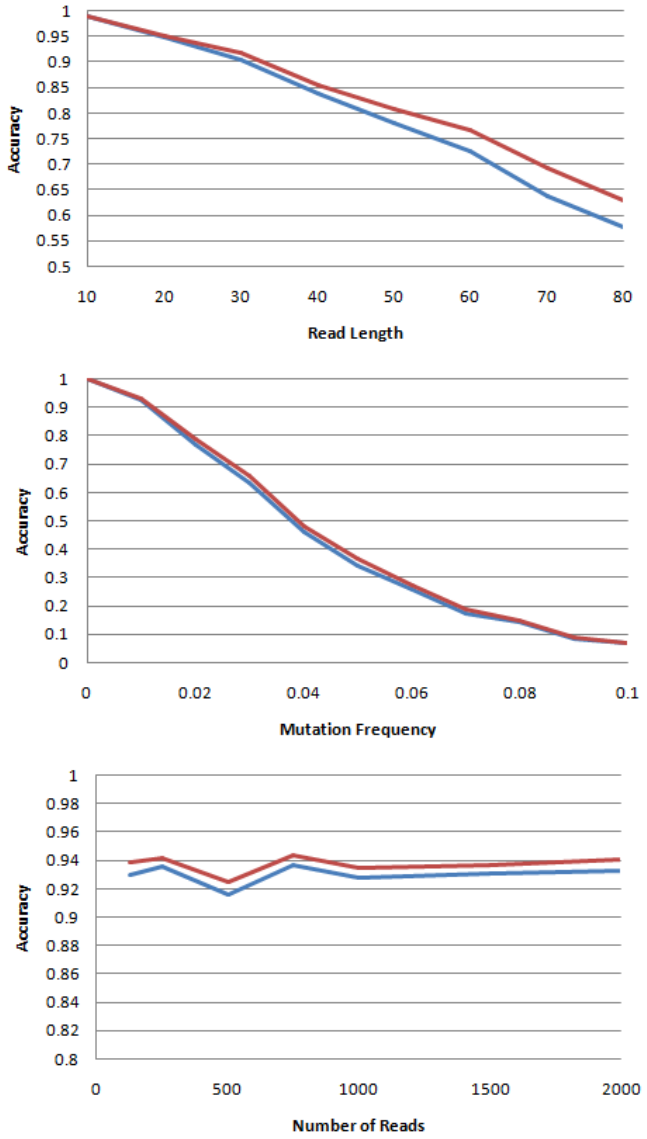

Figure 3. Comparison of accuracy for basic matching (blue) and iterative referencing (red) for varying read length, mutation frequency, and number of reads.

1) Testing variable mutation density: Unlike my simulated data, which inserted mutations into the genome with equal probability at every base, in reality errors in the genome are more densely distributed in some regions and less densely in others. I hypothesized that such a variable density of mutations might allow iterative referencing to perform more accurately at higher mutation rates by first matching reads in regions of low mutation density and then extending into regions with higher mutation density, correcting genome mutations as it goes. To test this hypothesis I modifed the simultation to insert mutations only into the middle section of the genome. However, preliminary testing showed no significant improvement in iterative referencing for this variable mutation density, so I did not pursue this hypothesis any further.

C. Introduction of Sequencing Errors

Up to this point in my experiments, reads were drawn directly from the mutated reference genome. In reality, sequencing is not completely accurate, and substitution errors occur with approximately 0.5-2.5% frequency [6]. To more accurately simulate sequencing data, I introduced substitution errors into simulated reads with small probability. As figure 4 shows, accuracy decreases roughly linearly for both basic matching and iterative referencing as the error frequency increases. Throughout the remainder of testing I used an error frequency of 0.01.

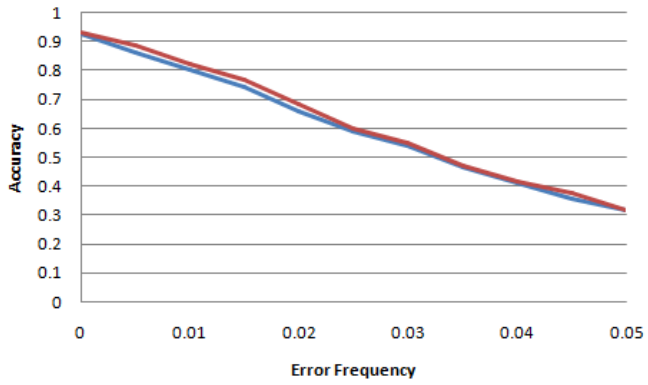

Figure 4. Comparison of accuracy for basic matching (blue) and iterative referencing (red) for varying frequency of sequencing errors.

D. Space Improvement in Iterative Referencing

One of the disadvantages of iterative referencing is that it requires a large amount of space. The algorithm must store every mutation found among matched reads in order to later search for common mutations. The total number of stored mutations can be represented as $M_{total} = M_{actual} + M_{errors}$ where M_{actual} is the number of true mutations in the reference genome, and M_{errors} is the number of sequencing errors among all the reads. Thus the total space requirement grows linearly with both the number of reads and the error rate, as shown in Figure 5. (Note that the space grows linearly only for small error frequencies, since at higher error frequencies fewer reads are successfully mapped, so fewer mutations are found.) Current generation DNA sequencers often sequence at a coverage depth of at least 15 reads [6], but this depth is constantly increasing as faster, higher-throughput sequencers

are developed. Thus, it is important to reduce the size of the mutations list if we can do so without compromising accuracy.

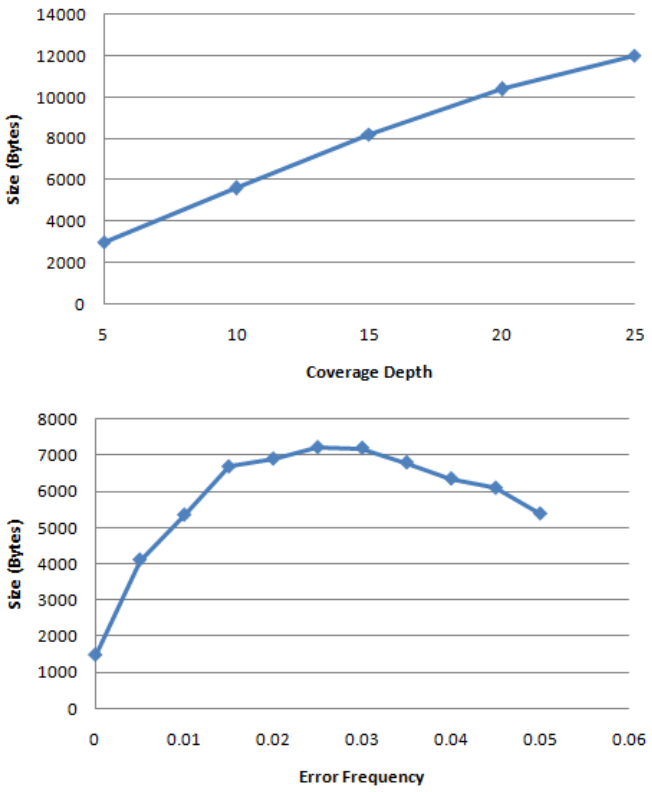

Figure 5. Space (in bytes) required to store the list of mutations in iterative referencing for varying depth of coverage and frequency of sequencing errors.

1) Simple Reduction: We can reduce the size of the mutations list by reducing the number of mutations that are saved. Notice that if we have a sequencing error rate of 1%, then on average at any base only 1 in 100 reads will contain an incorrect mutation. Thus, if we store the mutations from only 1% of reads we can expect to retain 99% of actual mutations to the reference genome, while retaining only 1% of incorrect mutations from sequencing errors.

To implement this method, we simply only save mutations from the first 1% of matched reads, which we call "contributing" reads. All other reads are matched to the genome, but their mutations are not added to the list, under the assumption that most of these mutations will either be redundant or will be sequencing errors.

It is noteworthy that this method does not reduce the growth rate of the space required to store mutations, since this space will still increase linearly with the number of reads. However, it should reduce the space by a factor of approximately 99%, which is a significant improvement.

2) Coverage Normalization: The simple reduction described above will only perform accurately if reads are evenly distributed so that coverage is consistent over the whole genome. In contrast, if the genome has highly-variable coverage, then regions with low coverage may not be represented in the 1% of contributing reads, so these regions may never be updated to reflect the actual genome. To correct this, we allow the first k matched reads that overlap a given base to contribute to the mutations for that base. For example, if $k = 1$ then for each base only the first matched read that overlaps that base can contribute a mutation for that base to the mutations list.

This method initially greatly increases the space required, since it requires us to store a coverage vector of the same length as the genome to count whether each base has been covered by a read yet. We can reduce this space by splitting the genome into larger chunks, rather than counting each base individually. For this method, we keep track of the number of bases in each chunk that have been covered by reads so far. As long as this number is less than some threshold, then we allowed reads that match in that chunk to contribute to the mutations list. Once the number of matched reads in a chunk surpasses our threshold, we no longer count mutations from matched reads in that chunk. This technique will ensure that the number of reads that contribute mutations is normalized over the whole genome, but assumes that locally (within chunks), coverage remains fairly consistent.

Note that with this method the space required to store mutations is no longer linearly dependent on the depth of coverage, since we will now store at most k mutations for every base in the genome (Figure 7).

Figures 6-9 and Table I show the accuracy and space for each of these variations on iterative referencing. While the simple reduction reduces the space requirement most drastically, this comes at the cost of reduced accuracy by about 2-3% on average. The coverage normalization method seems to preserve accuracy even for a chunk size of 50 or 100 bases. While this method does not appear to reduce the space as significantly, notice that as expected the space remains constant as read length and number of reads increase. In contrast, the original and reduced iterative referencing increase linearly with read length and number of reads. Therefore we would expect our coverage normalization method to require significantly less space for high depth of coverage.

Table I
COMPARISON OF ACCURACY FOR THE DIFFERENT ALGORITHM VARIATIONS AS READ LENGTH INCREASES. C INDICATES THE SIZE OF EACH CHUNK IN BASES, D INDICATES THE DEPTH OF CONTRIBUTING READS FOR EACH BASE.

Read Length	Original IR	Reduced (1%)	C=1, D=1	C=1, D=2	C=50, D=2
10	.992	.992	.991	.992	.992
20	.972	.970	.972	.975	.972
30	.924	.917	.929	.925	.924
40	.867	.866	.878	.870	.873
50	.839	.827	.832	.836	.843
60	.784	.766	.801	.790	.788
70	.721	.709	.722	.722	.721
80	.679	.637	.661	.664	.660

V. CONCLUSIONS

While my results so far are not sufficient to conclusively show that the coverage normalization is an improvement over the basic iterative referencing algorithm, they strongly suggest that it is possible to reduce the space required by iterative referencing without compromising accuracy. In order to conclusively quantify the advantages of coverage normalization, I will have to run more tests to more thoroughly analyze

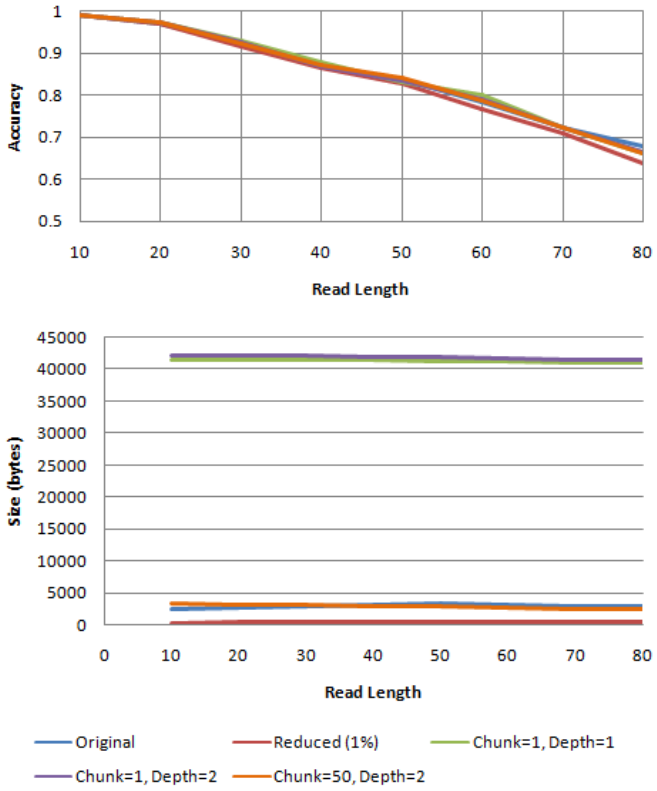

Figure 6. Comparison of accuracy and space requirements for the different algorithm variations as read length increases.

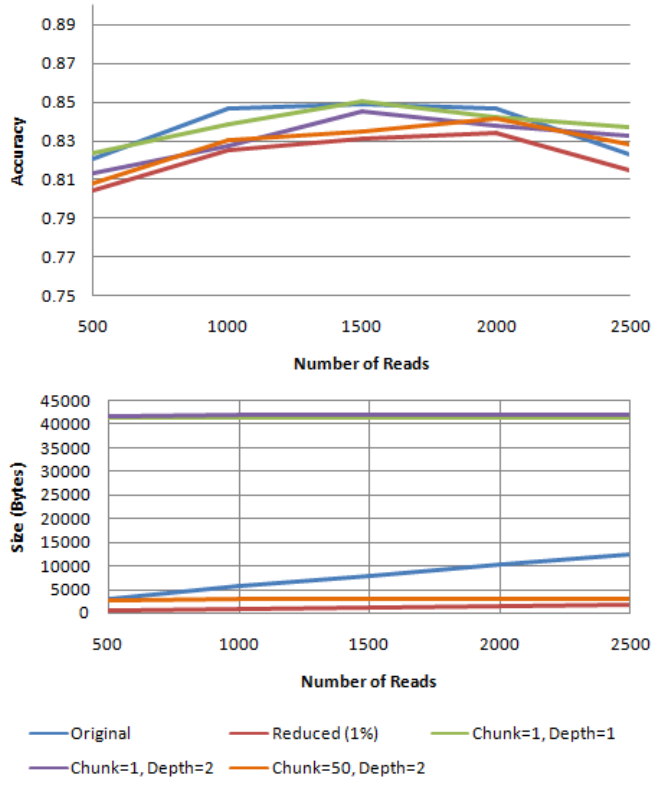

Figure 7. Comparison of accuracy and space requirements for the different algorithm variations as number of reads increases. Note that while the space required for the original iterative referencing and reduced algorithms increases with depth of coverage, the space for coverage normalization methods remains constant.

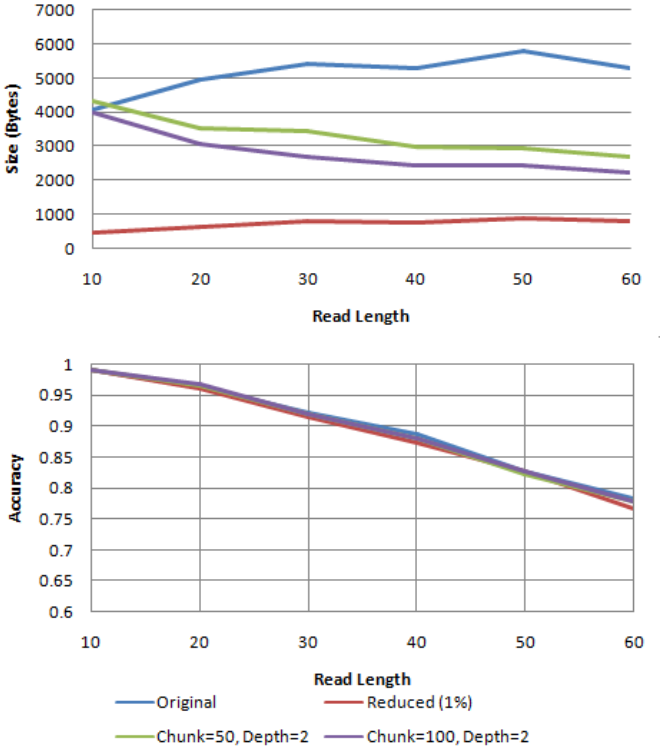

Figure 8. Comparison of accuracy and space requirements for the different algorithm variations as read length increases.

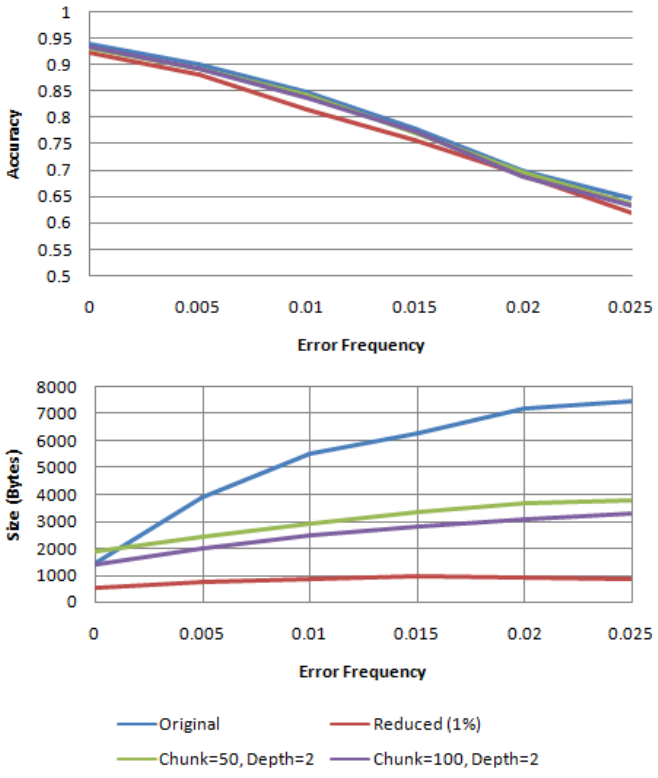

Figure 9. Comparison of accuracy and space requirements for the different algorithm variations as error frequency increases.

the accuracy and space requirements of the algorithm across different chunk sizes and depths.

Furthermore, simulated data can only roughly represent the actual biological data from gene mutations and sequencing. I will need to test these methods on actual genomic data to make sure that they continue to perform well. This would also allow me to incorporate the PHRED quality score into the iterative referencing algorithm, as described by Ghanayim and Geiger [2], to more accurately predict which read mutations are sequencing errors.

Finally, I would like to extend the algorithm to work for approximate matching with up to 2 or more errors. I believe that this would lead to both greater accuracy overall and a greater improvement in iterative referencing.

VI. Literature Cited

References

[1] M. Salson, T. Lecroq, M. Leonard, and L. Mouchard, "A Four-Stage Algorithm for Updating a Burrows-Wheeler Transform," *Theoretical Computer Science*, 2008.

[2] A. Ghanayim and D. Geiger, "Iterative Referencing for Improving the Interpretation of DNA Sequence Data," *Technion Technical Report*, 2013.

[3] M. Burrows and D. Wheeler, "A block sorting lossless data compression algorithm," *Technical Report 124*, 1994.

[4] B. Langmead, C. Trapnell, M. Pop, and S. L. Salzberg, "Ultrafast and memory-efficient alignment of short DNA sequences to the human genome," *Genome Biology*, 2009.

[5] M. Salson, T. Lecroq, M. Leonard, and L. Mouchard, "Dynamic Extended Suffix Arrays," *Journal of Discrete Algorithms*, 2009.

[6] D. Kelley, M. Schatz, and S. Salzberg, "Quake: quality-aware detection and correction of sequencing errors," *Genome Biology*, 2010.