Short-Read Genome Resequencer with SNP and Indel Detection

James Gomez

Department of Computer Science, University of California, Los Angeles

Abstract**—Computational genomics has played a vital role in the uncovering of genomic variation within species, allowing us to study its effect on disease and other phenotypical traits. Next-generation sequencing allows us to obtain short genome sequence data (reads) at a high throughput and low cost. Because of the computational challenges that next-generation sequence data pose to *De Novo* assembly, and because of the high availability of reference genomes, resequencing has emerged as an efficient alternative to assembling donor genomes and uncovering variants. This paper presents an implementation of a short-read resequencer, which uses a hashtable index of the reference genome to efficiently map individual reads to the reference in constant time. In addition, the resequencer is able to detect SNP and indel variants with reasonable accuracy. Performance is compared to a baseline implementation, which uses an exhaustive search approach.**

I. INTRODUCTION

Genomics continues to play a major role in furthering our understanding of gene function in living organisms. In medicine, for instance, the use of genome wide association studies (GWAS) has allowed for a deeper understanding of the genetic variants involved in the development of certain genetic diseases [1].

In order to be able to analyze genomic variation within species, we must first obtain the assembled genomes from individuals (donors) of the species. Previous methods utilized Sanger-based sequencers, which generate fragment sequences (reads) of several hundred base-pairs (bp) [2] [3] [4], and *de novo* assembly [5] to obtain complete or near complete genomes from the fragmented sequence data alone. However, the current state of the art of next-generation sequencers generate much shorter reads of about 30-100 bp, albeit at a dramatically increased throughput and much lower cost [3] [4]. Because of the computational challenges that these new short read data pose to *de novo* assembly [3] [5], and because of the high availability of already assembled reference genomes, many current bioinformatics efforts have focused on reference-based techniques rather than *de novo* assembly as a more computationally practical method of finding genomic variants.

Resequencing, or reference-based assembly, is the process of mapping reads to an already assembled reference sequence, thereby assembling a donor sequence with respect to it. Once the donor reads are mapped to the reference, we can compare the donor assembly to the reference in order to find variants in the donor.

There are many types of variants that occur in the genomes of individuals of a species. Two of the most important types of variants are single-nucleotide polymorphisms (SNPs), and small sequence insertions and deletions (indels). This is because of their influence on human traits and disease.

This paper outlines an implementation of a hashtable-based genome resequencer with both SNP and indel detection capabilities and analyzes its performance with respect to a naïve implementation.

II. PROBLEM FORMULATION

The goal of resequencing is to assemble the genome of a donor of interest with respect to a reference genome. This allows us to then identify the variants within the donor genome by comparing it to the reference.

In this implementation, we are given the assembled reference genome and a set of unmapped reads of 50 bp each from an unknown donor genome. Formally, we can reduce the biological problem of resequencing to the following two-step computational problem…

(1) *Given a reference sequence and a set of reads from an unknown donor sequence, assemble the donor sequence.*

Input:

- Reference sequence
- Set of reads from an unknown donor sequence

Output:

- Assembled donor sequence

(2) Given the reference sequence and assembled donor sequence from (1), find all SNP and indel variants in the donor with respect to the reference.

Input:

- Reference sequence
- Assembled donor sequence

Output:

- List of donor variants (SNPs and indels)

III. METHODOLOGY

When resequencing small genomes (thousands of base pairs) we can solve the problem in a reasonable amount of time using a simple exhaustive search of the reference genome for every read in our set of reads. This approach quickly becomes infeasible with respect to time, however, as the size of the genome increases to millions or billions of base-pairs.

One straightforward method for overcoming this challenge is to build a hashtable index of the reference genome and use it to map parts of the reads perfectly in constant time. The following section outlines a hashtable-based implementation as a time-efficient solution to the problem presented in section II, using the naïve exhaustive search implementation as a baseline for comparison. Subsection A outlines the generation of our test data. Subsections B through D solve problem (1) and subsection E provides a solution to problem (2).

A. Generation of Sample Genome Data

In order to be able to run the resequencer against sample data, a set of pre-generated genomes of varying length are provided along with corresponding reads and variant answer-keys. These data are stored as text files, which follow the FASTA format. The sample data were generated as follows.

First, a random reference genome is generated such that each base is chosen out of the four bases (A, C, G, T) with equal probability each time. Next, the reference is mutated at random positions to include SNPs and indels. Indels range from 1 to 5 bp for simplicity. The resultant mutated genome is the donor. From the donor genome, reads of length 50 bp are randomly sampled such that we achieve 30x coverage (each position in the genome is sampled an average of 30 times). As they are sampled, single-base error and whole-read error (garbage reads) are introduced into the reads at a rate of 0.01 and 0.05, respectively. Finally, an answer-key containing the location and quality of each of the donor variants (with respect to the reference) is generated.

B. Baseline Alignment

The baseline algorithm is very straightforward. For every read in our set of reads, we slide the read along every position in the reference, comparing the bases at each position in the read with the bases in the reference in the current position and saving the position with the least mismatches. Because we know that variants in the donor are relatively sparse, we can use a threshold to weed out garbage reads from our set of reads. Thus, we save the position in the reference with the least mismatches as long as it is below our threshold. Otherwise, the read is considered a garbage read and no position is saved. The algorithm is given in Algorithm 1 below.

```
NAIVE_ALIGNMENT(ref, reads, thresh):
   alignments = []
   for r in reads:
      best_pos = None
      min = ∞
      for i=0 to ref.length-r.length+1:
         sum = 0
         for j=0 to r.length:
            if ref[i+j] != r[j]:
               sum += 1
         if sum <= thresh and sum < min:
            min = sum
            best_pos = i
      alignments[r].append(best_pos)
   return alignments
```

Algorithm 1: Naive alignment of donor reads

Although it offers constant space-complexity and simplicity of implementation, the naïve alignment solution has extremely poor time performance for any decently sized genome. If we let n, m, and l be the length of the reference genome, the number of reads in our set of reads, and the length of each read, respectively, then it's time complexity is given as $O(n * m * l)$, which is very poor for a large n and correspondingly large m of 30x coverage.

C. Hashtable Index Alignment

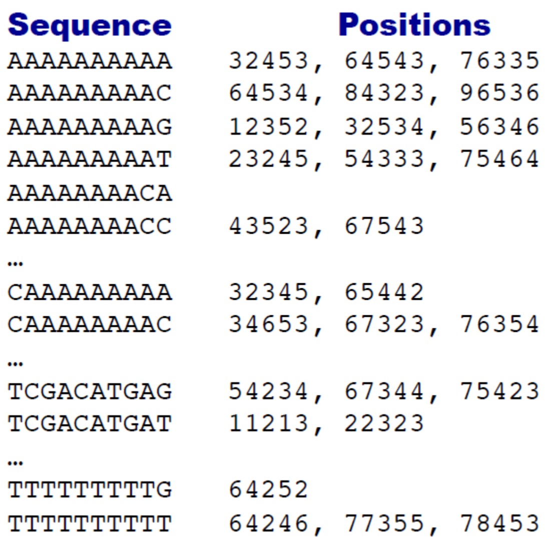
i. Building an Index for the Reference Genome

The hashtable alignment algorithm requires a preprocessing step to build the index of k'mers for the reference genome. Here, this is accomplished in a subroutine, shown in Algorithm 2, which takes the reference string and a k value as parameters. We slide across each position in the reference, saving unique substrings of length k as our keys and a list of the positions in which that substring occurs as our values. This can be done in $O(n)$ time, where n is, again, the length of the reference genome.

```
CREATE_INDEX(ref, k):
    index = {}
    for i=0 to ref.length-k+1:
        kmer = ref[i: i+k]
        index[kmer].append(i)
    return index
```

Algorithm 2: Subroutine for building the hashtable index of k'mers from the reference genome

When generating an index, it is important to choose a k value which provides good unique substring mapping (minimal collisions) while also keeping the key as short as possible so as to minimize space consumption. In this implementation, k is chosen to be 10 since it evenly divides into the read length of 50 bp and has empirically shown to have a good balance between mapping quality and space consumption. Figure 1 shows an illustration of an index with sequence k'mers of length 10 as keys.



Sequence	Positions
AAAAAAAAAA	32453, 64543, 76335
AAAAAAAAAC	64534, 84323, 96536
AAAAAAAAAG	12352, 32534, 56346
AAAAAAAAAT	23245, 54333, 75464
AAAAAAAACA	
AAAAAAAACC	43523, 67543
…	
CAAAAAAAAA	32345, 65442
CAAAAAAAAC	34653, 67323, 76354
…	
TCGACATGAG	54234, 67344, 75423
TCGACATGAT	11213, 22323
…	
TTTTTTTTTG	64252
TTTTTTTTTT	64246, 77355, 78453

Figure 1: An index containing k'mers as keys and a list of positions in the reference sequence as the values [6]

ii. Fast Alignment Using a K'mer Index

Once the reference index has been built, we can use it to map our reads to the reference in constant time. Mapping with the index proceeds as follows.

For every read in our set of reads, we break up the read into non-overlapping k'mers of the same length as the keys in our index. For every one of our k'mer subsequences, we test to see if there is a perfect match within our index. If there is a match, we obtain the corresponding list of positions in which the k'mer occurs in the reference from the index and compare the entire read to the reference at those positions. We keep the first mapping with the lowest number of total mismatches between the read and the reference and store that position as the alignment for that read.

```
HASH_ALIGNMENT(ref, reads, index, k):
    alignments = {}
    for r in reads:
        best_pos = None
        min = ∞
        for i=0 to r.length, i+=k:
            kmer = r[i: i+k]
            positions = index[kmer]
            for p in positions:
                sum = 0
                for j=0 to r.length:
                    if r[j] != ref[p+j-i]:
                        sum += 1
                if sum < min:
                    min = sum
                    best_pos = p-i
        alignments[r].append(best_pos)
    return alignments
```

Algorithm 3: Hashtable alignment of donor reads

Compared to the baseline algorithm, using the hashtable index approach performs drastically better in terms of time. This is because it can map individual reads in constant time via lookup, whereas the baseline must scan the reference linearly to find the best match for every read. Thus, the time complexity for alignment (including preprocessing for the hashtable approach) is reduced from $O(n*m*l)$ to $O(n+m*l)$, where n, m, and l are the length of the reference genome, the number of reads in our set of reads, and the length of each read, respectively.

However, this approach can be become very expensive in terms of memory for large genomes since the index for the reference must be kept in main memory to leverage its high performance lookup. Many approaches have been developed in recent years to overcome the memory limitations of pure hashtable based aligners. Examples include BWA [7] and Bowtie [8], which use the Burrows-Wheeler Transform to index the reference with high space efficiency. These, however, are outside of the scope of this implementation.

D. Generating a consensus donor genome

After we've aligned our set of reads to the reference genome, we need to infer the most likely donor genome from our aligned reads by consolidating the overlapping reads into a single sequence.

The first step is to generate a pile-up of our reads. Here, we accomplish this by creating an array of integers which keeps track of the count of each base that is reflected in the aligned reads at each position. For each of our reads, we update the base counts at the positions reflected by the reads. A simple pile-up of reads is depicted in Figure 2.

```
Reference:  TCCACATGAGATCTGTAGAGCTGTGAGATC
Reads:     | TCGACATGAGATCGGTAGAACCGT
           |    GACAAGAGATCGGTAGAGCCGTGA
           |          TGAGATCGGTAGAGCCGTGAGATC
```

Figure 2: A small reads pile-up. SNPs are highlighted in red and read errors are highlighted in blue [6].

Once we've generated our pile-up array, we loop through each position and use a simple majority vote (greater than 50%) to resolve base discrepancies. We write the winning base at each position to a string, which represents the consensus donor. If there is a tie or there is no simple majority for any base, we fall back to the base in the reference genome.

Given that we know that the error rate from next-generation sequencers is relatively low (0.01 in the sample genomes) and that we have high coverage (30x in the sample genomes), we can expect to have less than one base discrepancy per position due to single base sequencing error. This makes our simple voting algorithm robust to single base errors. In general, the overwhelming consensus of the bases at a single position will overrule any single base errors reflected in the reads.

E. Finding Variants

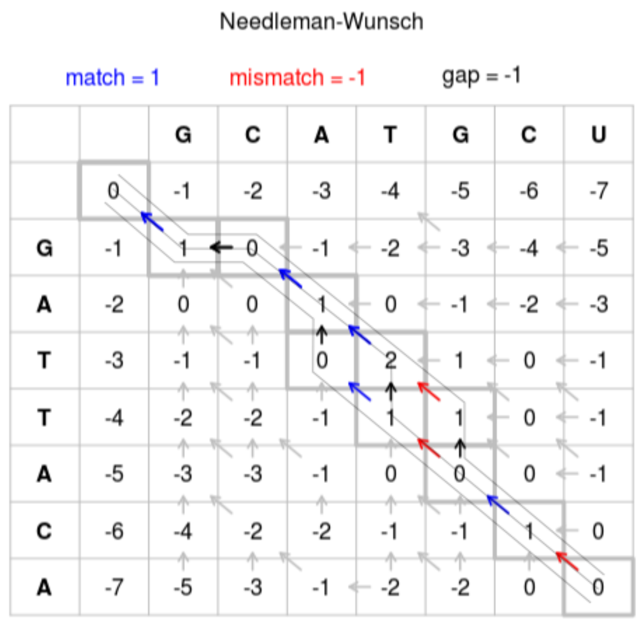
iii. SNPs

Finding SNPs in the donor genome is very straightforward. We loop through our donor genome and inspect batch sequences of 100 bp. For each of our batches, we compare the base at every position with the base in the corresponding position in the reference genome. If the bases do not match, we save the position in the reference in which the mismatch occurs along with the original and mutated bases. We also keep track of the amount of mismatches that we encounter in every batch. If there are no mismatches, we simply continue to our next batch of bases. If we find one or two non-consecutive mismatches in our batch, we write the mismatches to our variants answer key as SNPs. Otherwise, we must consider the possibility of indels within our batch.

iv. Indels and edit-distance alignment

Detecting indels is a much more complex process. In order to find the indels, we must find the gaps in both the reference and donor sequences that produce the most optimal alignment. Using an exhaustive search approach would yield exponential time performance.

To do so in quadratic time, we use a set of dynamic programming algorithms to perform alignment of two sequences of interest: The Needleman-Wunsch algorithm for performing global alignment, the Smith-Waterman algorithm for performing local alignment, and an affine-gap variant of Needleman-Wunsch. These algorithms work by keeping score of the amount of change needed to transform one subsequence to the other subsequence. We keep track using a matrix which stores the scores at the current corresponding sequence locations. A separate matrix can be used to keep track of trace-back pointers, which can be followed after the matrix is completely filled to construct the optimal alignment between the two strings. An example Needleman-Wunsch alignment is shown in Figure 3.



		G	C	A	T	G	C	U
	0	-1	-2	-3	-4	-5	-6	-7
G	-1	1	0	-1	-2	-3	-4	-5
A	-2	0	0	1	0	-1	-2	-3
T	-3	-1	-1	0	2	1	0	-1
T	-4	-2	-2	-1	1	1	0	-1
A	-5	-3	-3	-1	0	0	0	-1
C	-6	-4	-2	-2	-1	-1	1	0
A	-7	-5	-3	-1	-2	-2	0	0

Figure 3: Neeedleman-Wunsch global alignment scoring matrix with trace-back pointers.

When it is determined that we must search for indels in our current reference and donor batch, as explained in the previous subsection, we perform one of aforementioned dynamic programming alignments with our current batch subsequences. This implementation allows the user to specify which of the alignment methods to use at the command line. The alignment proceeds from the top-left corner of the matrix and expands row-wise until the matrix is completely filled with scores. We can obtain the final alignment score from the bottom-right most matrix element.

```
ACAGTTAGAT---CTATTGTCA
ATAG--AGATTTACTATTGCCA
```

Figure 4: Alignment of two sequences after running an edit-distance alignment. Insertions are in green, deletions are in red, and SNPs are in blue.

To construct the alignment of the two sequences, we start at the bottom-right matrix element and follow the trace-back pointers back to the top-left. An up move corresponds to a gap for sequence 1 and a right move corresponds to a gap for sequence 2 (or vice-versa depending on the matrix layout). Diagonal moves correspond to either matches or SNPs. Once we've finished our trace-back, our result will look something similar to Figure 4, where gaps are depicted as dashes. We can then scan the resultant alignment to determine where the SNPs and indels occur in the current batch. Inserts will correspond to consecutive bases in the donor whose corresponding locations in the reference are dashes, and deletions will correspond to consecutive bases in the reference whose corresponding locations in the donor are dashes.

Using the dynamic programming approach, we can uncover indels and SNPs with reasonable accuracy. The approach falls short when there are SNPs bunched very close together, as this can easily be confused with an insertion or deletion. Shorter insertions and deletions are also harder to accurately detect than longer ones, resulting in a fair amount of false positives for insertions and deletions of less than 3 bp. In general, the affine-gap alignment approach was the most accurate of the three alignment strategies with the fewest false-positives.

IV. RESULTS

Table 1 shows the results of running the baseline alignment and hashtable alignment algorithms for variously sized reference and donor genomes on a Windows 8.1 laptop with an Intel Core i7 CPU and 8 GB of RAM. The alignment script is written in standard Python 2.7 with NumPy support. In all cases, the hashtable approach performs drastically better than the baseline. Because the baseline algorithm very quickly becomes infeasible as the size of the genome increases, only runtimes for 10,000 bp and 100,000 bp were obtained for the baseline algorithm.

Genome size (bp)	Baseline (sec)	With Hashtable (sec)
10,000	334.617	0.379
100,000	33,946.738	2.749
1,000,000	n/a	49.185
10,000,000	n/a	2636.639
100,000,000	n/a	~ 3 days

Table 1: Total Alignment times for the baseline alignment and hashtable alignment algorithms.

V. CONCLUSION

Resequencing continues to play a key part in efficiently uncovering genomic variation within species, thus paving the way for understanding the ways in which these variations affect outward traits and disease. This paper has presented a resequencer implementation with a hashtable-based aligner and reasonably accurate SNP and indel detection capabilities. Compared to the baseline approach, the hashtable alignment performs drastically better at the cost of increased memory consumption proportional to the size of the reference genome. Given enough memory resources, the hashtable alignment approach is a good choice for quickly resequencing both small and large genomes. More sophisticated approaches have been implemented, however, to drastically decrease memory consumption using novel indexing structures.

VI. REFERENCES

[1] E. Eskin, "Discovering Genes Involved in Disease and the Mystery of Missing Heritability," University of California, Los Angeles.

[2] F. Sanger, S. Nicklen and A. R. Coulson, "DNA sequencing with chain-terminating inhibitors," *Biochemistry*, vol. 74, no. 12, pp. 5463-5467, 1977.

[3] M. Pop and S. L. Salzberg, "Bioinformatics challenges of new sequencing technology," *Trends in Genetics*, vol. 24, no. 3, pp. 142-149, 2008.

[4] O. Morozova and M. A. Marra, "Applications of next-generation sequencing technologies in functional genomics," *Genomics*, pp. 255-264, 2008.

[5] K. Paszkiewicz and D. J. Studholme, "De novo assembly of short sequence reads," *Briefings in Bioinformatics*, 2010.

[6] E. Eskin, *Computational Genetics Lecture 1: Introduction to Computational Genetics and the HapMap*, University of California, Los Angeles, Spring 2014.

[7] H. Li and R. Durbin, "Fast and accurrate short read alignment with Burrow-Wheeler transform," *Bioinformatics,* vol. 25, no. 14, pp. 1754-1760, 2009.

[8] B. Langmead, C. Trapnell, T. Pop and S. L. Salzberg, "Ultrafast and memory-efficient alignment of short DNA sequences to the human genome," *Genome Biology*, vol. 10, no. 3, p. R25, 2009.