A LOW-COMPLEXITY PROBABILISTIC GENOME ASSEMBLY BASED ON HASHING FUNCTIONS WITH SNP DETECTION

Naji Mounsef,[1] *Lina Karam,*[1,*] *Zoé Lacroix,*[1,2] *and Christophe Legendre*[1]

[1]Department of Electrical Engineering, Arizona State University
Tempe AZ 85287, USA
[2]Pharmaceutical Genomics Division, Translational Genomics Research Institute (TGen)
13400 E Shea Blvd, Scottsdale AZ 85259, USA
[*]Corresponding Author: karam@asu.edu

ABSTRACT

This paper presents an efficient low-complexity genome assembly algorithm with the ability to detect bit errors (SNPs). A hashing function is used to reduce the complexity of the assembly process. The algorithm is tested against genomic sequences of different lengths. Its performance in terms of completeness, accuracy, and efficiency (time and space) is evaluated against Phrap, a well-known sequence assembly tool. It is shown that the proposed assembly algorithm outperforms Phrap in terms of accuracy, time, and memory.

1. INTRODUCTION

Each cell found in any living organism contains chromosomes made of sequences of DNA base pairs composing a genome, which denotes a set of instructions that directs the replication and function of each organism [12]. The information coded on each genome sequence is essential for medical, agricultural, and several other research areas [3]. However sequencing machines can handle only 500-1000 base fragments at a time therefore thousands of sequences are usually produced to cover the whole genome.

Genome sequencing consists of three main stages: sequencing, assembly, and finishing. In the sequencing stage, a DNA from a source is fractured into thousands of small pieces. These pieces are then "read" by an automated DNA sequencer machine. In the assembly stage, complex computer algorithms operated by a human assembly team assemble these short sequences back together into a partially complete 'draft' genome sequence. Finally, human 'finishers' curate the assembly to correct sequencing, and run additional sequencing reactions to fill in the un-sequenced gaps.

A genome assembly algorithm takes all the DNA sequences and aligns them to one another, detecting all places where two of the short sequences, or *reads*, overlap. These overlapping reads can be merged together, and the process continues. The main idea is that two overlapping reads - a suffix of one is a prefix of another – most probably originate from the same region of the genome and can be assembled together. For example, suppose that the following sequences need to be assembled: {CTAA, ATGA, GCTA, TGAT, AGCT}. The reassembled sequences TGATGAGCTAA, AGCTAATGAT, CTAAATGAGCTATGATAGCT are all composed of the given sequences, but one is of length 11, the second is of length 10, and the third is of length 20. The assembly algorithm usually looks for the shortest string *S* such that every given sequence is a substring of *S*. This problem has been shown to be NP-hard [12]. To solve this problem, a greedy approach is used in software such as TIGR Assembler [14], Phrap [5], and CAP3 [7]. Other approaches exploiting techniques developed in the field of graph theory - representing the sequence reads as graph nodes [8] or as edges [2] – can be more efficient because the assembler can find Eulerian paths in linear time while the problems associated with the overlap-layout-consensus paradigm are NP-complete [8]. The AMOS Comparative Assembler [13] follows a fundamentally different approach: the overlap step is skipped entirely. Instead, reads are aligned to a reference genome (template) using a modified version of the MUMmer algorithm [9].

The DNA sequences that need to be assembled may contain errors such as single nucleotide polymorphisms (SNPs) which may result in incorrectly assembled genome. SNPs are DNA sequence variations occurring when a single nucleotide (A, T, C, or G) in the genome sequence is changed [15]. In other words, it is a single base pair difference between two sequences. For example, a SNP might alter the DNA sequence A**A**TCGTA to A**T**TCGTA. When reassembling the genome, SNPs should be taken into consideration hence the two sequences AATCGTA and ATTCGTA should be identified as one sequence. Otherwise the resulting sequence is wrongly assembled.

Different approaches to detect SNPs have been proposed. They include TRACE-DIFF [4], PolyPhred [10], pta and AGENT[1], TGICL [11], and autoSNP [1]. The ma-

This research was partially supported by the National Science Foundation (grants IIS 0431174, IIS 0551444, and IIS 0612273). Any opinion, finding, and conclusion or recommendation expressed in this material are those of the authors and do not necessarily reflect the views of the National Science Foundation.

[1]The Paracel Transcript Assembler (PTA) is available at

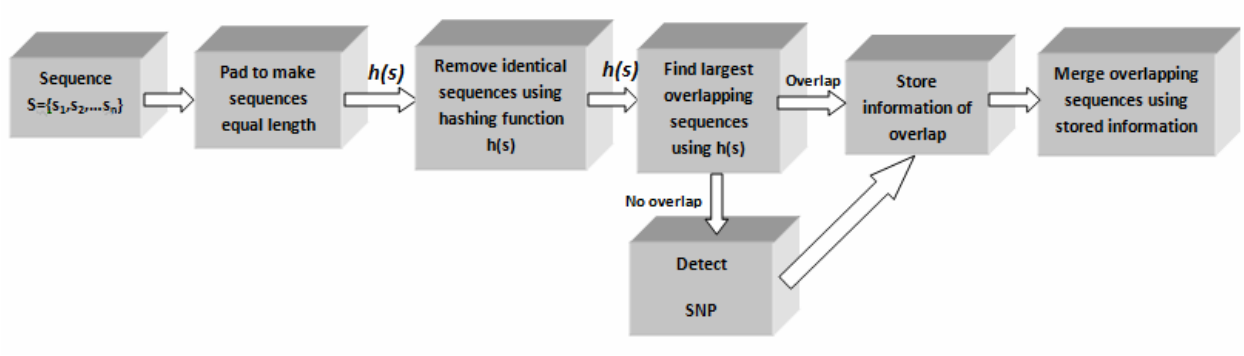

Figure 1. Proposed Algorithm Block Diagram

jor shortcoming of these approaches is the fact that potential SNPs positions are determined from assemblies that align all available sequences together, without taking into consideration if they contain differing SNP positions or if they come from different organisms.

This paper proposes an efficient low-complexity assembly algorithm with the ability to detect bit errors (SNPs) whose performance is compared against Phrap [5], a well known assembly tool. Section 2 presents a description of the proposed assembly algorithm. Results using the known genome for Fugu are reported in Section 3 to illustrate the performance of the proposed scheme. Finally, a conclusion is given in Section 4.

2. PROPOSED GENE ASSEMBLY ALGORITHM

The genome assembly problem takes as input a set of strings $S = \{s_i, \ldots, s_N\}$, where $s_1, s_2, \ldots, s_N$ have different lengths. It outputs the shortest string s that contains all the input strings s_i as substrings.

Figure 1 shows a block diagram of the proposed gene assembly scheme based on hashing functions. The proposed method has the ability of aligning and assembling sequences of different lengths into a genome, with low computational complexity and high accuracy, and is also capable to detect SNPs and use them in the assembling process.

A hash function maps a set of values (e.g., a sequence of strings, objects, numbers) into one single value, usually an integer or some other numeric value. This value, referred to as a *hash value*, is used to locate a position within a hash table or a special kind of array [6].

Let $S = \{s_1, \ldots, , s_N\}$ be a set of N string sequences of respective different lengths $l_1, \ldots, l_N$ and let $L = max\{l_1, \ldots, l_N\}$. Suppose that $s_i = \{m_{i,1}, \ldots, m_{i,l_i}\}$ where $m_{i,j}$ is a numeric value corresponding to the j^{th} alphabet letter forming s_i. All strings s_i are padded so that $l_i = L$ resulting in a $N \times L$ matrix S where the string s_i forms the i^{th} row of S. Let $R_L = \{r_1, \ldots, r_L\}$ be a row vector of uniformly distributed random numbers of length L. The hashing function is then given by

$$H(S) = S.{R_L}^T \tag{1}$$

http://www.becabioinfo.org/bbrweb/cms/front_content.php?idcat=56.

Instead of comparing each of the L letters of the N sequences s_i the method only compares N real numbers. This hashing method is used in the proposed scheme to find both similar sequences and the overlapping sequences. The resulting H is a vector of real numbers of length N.

In order to reduce the amount of sequences to compare in later stages, a pre-processing step is required to eliminate identical sequences. This is usually done by sorting the sequences in alphabetical order and working on all columns of the sequences. This is an L-dimensional problem, where L is the maximum length of the sequences. Using the proposed hashing function $H(S)$ given by (1) this process is reduced to a $1-d$ problem, which consists of comparing just one column of real numbers.

The next step where genome assembly algorithms typically spend the most time consists of the identification of the largest overlaps among the sequences. This step is computationally exhaustive because it consists in matching every pair with every possible overlap. The proposed hashing function $H(s)$ of (1) was used again to simplify this step. Because all the identical segments were removed, the longest overlap would be of length $L-1$. The $L-1$ rightmost of each sequence is compared with the $L-1$ leftmost of the rest of the sequences by using $H(s)$. Once this step is achived, the next overlap to be looked at is $L-2$ and so on. The process is repeated until the overlap is 1.

For each overlap, if no exact match is found, instead of going to the next overlap, a SNP detection algorithm is run to find if there are two overlapping sequences with almost exact overlap. This is done by computing the distance between the two sequences as a metric that captures the similarity between two sequences. In other words, the distance D between two sequences $p = \{p_1, p_2, \ldots, p_N\}$ and $q = \{q_1, q_2, \ldots, q_N\}$ is expressed by $D(p,q) = \sum_{i=0}^{N} sign \mid p_i - q_i \mid$ where $sign(x) = 0$ if $x = 0$ and $sign(x) = 1$ otherwise.

If $D(p,q) = 1$ then the two sequences differ by only one character, which is equivalent to having a SNP. Two sequences that only differ by one character are found to be overlapping. This guarantees that the bit positions of the SNPs are taken into consideration. However, instead

Table 1. Performance and comparison results of proposed Genome Assembly scheme

Original Length M	Number of Split Sequences	Number of SNPs	Average Length of Split Sequences	Min-Max of Length Sequences	assembled sequence		% matching		time(s)	
					proposed	Phrap	proposed	Phrap	proposed	Phrap
20,789	456	25	200	20-2000	20,789	20,846	100%	-	4.07	4.48
23,456	102	20	689	70-7000	23,456	23,481	99.99%	-	1.65	3.51
100,000	380	30	500	50-5000	99,465	101,093	-	-	22.082	33.43
1,000,000	7,710	50	500	50-5000	989,874	*Out of memory*	-	-	48.032	-

of calculating the difference between all the bits, the metric was modified to calculate only the difference of each two bits (p_i, q_i) for each sequence until its value becomes larger than a threshold value, which represents the maximum number of SNPs allowed in one sequence and depends on the tolerance of bit errors determined by the user. According to [15], a single SNP may occur every 500-600 bits; hence, a threshold of 1 was used to determine the presence of a SNP as the average length of the sequences used for simulation is also around 500.

To combine sequences, every time two sequences are found to exhibit the largest overlap, only the indices of the two overlapping sequences and the position of the overlap are stored instead of storing the sequences. Once the whole algorithm has finished looking for overlaps, the sequences are combined into the resulting genome using the stored indices and overlap-position information.

3. RESULTS

The proposed algorithm was tested against the fourth Fugu genome retrieved from UCSC[2] assembled by the US DOE Joint Genome Institute (JGI)[3]. In order to test its performance on sequences that contain SNPs (bit errors), each test dataset was generated by splitting randomly and several times a sub-sequence of length M of the Fugu genome into sub-sequences of different lengths. The SNPs were created by changing randomly selected bits in some sequences using an error probability pe of values ranging from 1% to 20%. The distribution of the test datasets with respect to M is similar in size and in average sequence length to the genomic Fugu sequences retrieved from TraceDB at NCBI[4] which has an average length of 1,081, a minimum length of 72, and maximum length of 12,001. The accuracy is assessed by the output cardinality (how many sequences were produced by the approach), the output length, and the percentage of identity with respect to the genome.

An accurate result is a single sequence of length at most M and similar to the genome. Time and space capture the efficiency of the approach. Because of Matlab memory limitations, the proposed scheme and Phrap [5] were tested for M as low as 10 characters up to 1,000,000 characters (the whole Fugu genome length is 400,509,340). Matlab was used to run the simulations of the algorithm on a 32-bit Intel core duo 1.5 GHz processor. The results are summarized in Table 1.

The results show that the proposed scheme produces a single output sequence of length at most M in all cases and was able to recognize SNPs and merge the sequences that differ by a SNP. In contrast, in all tested cases, Phrap fails to reassemble the input dataset into a single sequence and results in several assembled fragments; this is probably due to the fact that an overlap has to be of length at least 32 to be considered by Phrap, whereas the proposed scheme does not have such a limitation. For Phrap [5], the length of the longest fragment (maximum length) is reported in Table 1 along with the sum of the length of all the assembled fragments which represents the total length of the reassembled sequence. The assembled sequences of size M produced by the proposed algorithm match exactly with the original sequence (100% match). Once the Matlab limitations have been addressed and the algorithm tested for the complete genome, the evaluation of the scientific accuracy of assembled sequences shorter than M will be evaluated. Therefore, in Table 1, the percentage of matching is only reported when the resulting assembled sequence has the same length M as the original genome sub-sequence. Nevertheless the results show that the proposed scheme is more accurate and more efficient (faster) than Phrap.

Moreover, despite the Matlab memory limitations, the proposed scheme performed well on the test dataset generated from an input of length 1,000,000 while Phrap ran out of memory. The good efficiency results of the proposed scheme are also due to the fact that only the indices of overlapping sequences and the index of the overlap position are stored, which is an efficient way of using memory. Hence, as shown in Table 1, the proposed scheme outperformed Phrap in terms of accuracy, speed, and memory.

4. CONCLUSION

In this paper, an efficient genome assembly algorithm that uses a hashing function was proposed, tested, and compared to Phrap, a well known assembly tool [5]. It was shown that the algorithm has the ability of reassembling correctly almost all of the given sequences even in the presence of SNPs in a relatively very short amount of time.

Future works include the implementation of the method in C code to be used as a standalone service and tested against the genomic Fugu sequences retrieved from

[2]The Fugu genome used for this test is available at http://hgdownload.cse.ucsc.edu/downloads.html#fugu.

[3]For more information on the Fugu genome, see the JGI project website at http://genome.jgi-psf.org/Takru4/Takru4.info.html and http://www.fugu-sg.org/.

[4]The Fugu sequences were downloaded from ftp://ftp.ncbi.nih.gov/pub/TraceDB/takifugu_rubripes.

NCBI's TraceDB and compare with the four different versions of the Fugu genome published at US DOE Joint Genome Institute (JGI).

5. REFERENCES

[1] G. Barker, J. Batley, H. O' Sullivan, K. J. Edwards, and D. Edwards. Redundancy based detection of sequence polymorphisms in expressed sequence tag data using autoSNP. *Bioinformatics*, 19(3):421–422, 2003.

[2] S. Batzoglou, D. B. Jaffe, K. Stanley, J. Butler, S. Gnerre, E. Mauceli, B. Berger, J. P. Mesirov, and E. S. Lander. ARACHNE: A Whole-Genome Shotgun Assembler. *Genome Res.*, 12(1):177–189, 2002.

[3] P. A. Benjamin. *Genetics - A Conceptual Approach*. W.H. Freeman and Company, 2005.

[4] J.K. Bonfield, C. Rada, and R. Staden. Automated detection of point mutations using fluorescent sequence trace subtraction. *Nucl. Acids Res.*, 26(14):3404–3409, 1998.

[5] B. Ewing and P. Green. Base-Calling of Automated Sequencer Traces Using Phred. II. Error Probabilities. *Genome Res.*, 8(3):186–194, 1998.

[6] S. Harris and J. Ross. *Beginning Algorithms*. Wiley-VCH, 2005.

[7] X. Huang and A. Madan. CAP3: A DNA Sequence Assembly Program. *Genome Res.*, 9(9):868–877, 1999.

[8] J.D. Kececioglu and E.W. Myers. Combinatorial Algorithms for DNA Sequence Assembly. *Algorithmica*, 13:7–51, 1995.

[9] S. Kurtz, A. Phillippy, A. Delcher, M. Smoot, M. Shumway, C. Antonescu, and S. Salzberg. Versatile and open software for comparing large genomes. *Genome Biology*, 5(2):R12, 2004.

[10] D. A. Nickerson, S. L. Taylor, and M. J. Rieder. Identifying Single Nucleotide Polymorphisms (SNPs) in Human Candidate Genes. In *Research abstracts from the DOE Human Genome Program Contractor-Grantee Workshop VIII - Bioinformatics Section*, Santa Fe, NM, February 27-March 2 2000.

[11] G. Pertea, X. Huang, F. Liang, V. Antonescu, R. Sultana, S. Karamycheva, Y. Lee, J. White, F. Cheung, B. Parvizi, J. Tsai, and J. Quackenbush. TIGR Gene Indices clustering tools (TGICL): a software system for fast clustering of large EST datasets. *Bioinformatics*, 19(5):651–652, 2003.

[12] M. Pop, A. Phillippy, A. L. Delcher, and S. L. Salzberg. Comparative genome assembly. *Briefings in Bioinformatics*, 5(3):237–248, 2004.

[13] M. Pop, S. L. Salzberg, and M. Shumway. Genome sequence assembly: algorithms and issues. *Computer*, 35(7):47–54, 2002.

[14] G.G. Sutton. TIGR assembler: a new tool for assembling large shotgun sequencing projects. *Genome Science and Technology*, 1:9–19, 1995.

[15] Z. Wang and J. Moult. SNPs, protein structure, and disease. *Human Mutation*, 17(4):263–270, 2001.