De Novo Draft Genome Assembly Using Fuzzy K-mers

John Healy
Department of Computing & Mathematics
Galway-Mayo Institute of Technology
Ireland
e-mail: john.healy@gmit.ie

Desmond Chambers
Department of Information Technology
National University of Ireland Galway
Ireland
e-mail: des.chambers@nuigalway.ie

Abstract- Although second generation sequencing technology can be used to rapidly sequence an entire genome, assembly algorithms require a high level of coverage to produce a complete genomic sequence. We describe a fuzzy *k*-mer approach that is capable of rapidly ordering and orientating low coverage sequence reads with a high level of accuracy. Using this approach, a draft genome of *Mycoplasma genitalium*, sampled at varying low levels of coverage, was accurately anchored against the genome of *Mycoplasma pneumoniae*. The anchored reads were assembled into scaffolds with a vastly increased N50 length and an error rate of <1.5%.

Keywords- genome assembly, anchoring, fuzzy k-mers, fuzzy hash maps

I. INTRODUCTION

The rapid evolution of genome sequencing technologies in recent years has lead to a reappraisal of sequencing alignment and assembly strategies [1-3]. Using massive parallelism, second generation sequencing (SGS) technologies are capable of rapidly producing a very large number of short reads [3-8]. These twin characteristics of read number and read length have resulted in a move away from assembly strategies based on the traditional overlap graph [9] to more *k*-mer centric approaches, such as sequence graphs and de Bruijn graphs [10-12].

Regardless of the sequencing technology employed, the assembly of a set of sequence reads into a complete or draft genome is predicated on a sufficient number of overlapping reads being made available to an assembler. The level of overlaps, or coverage, is a function of the amount of oversampling employed during sequencing. To create a set of read fragments that represents 99.9% of a genome, an eight-fold (8X) level of coverage is required [13, 14]. Notwithstanding this and the recent rapid advances in DNA sequencing technology, many of the published genomes available in public repositories are of draft quality, at coverage levels as low as 2X [15]. Despite an acceleration in the number of completed genomes, the sheer number of potential candidate species available implies that most will either never be sequenced, or will be sequenced to draft quality only [16]. There is thus an evident niche for applications that are capable of generating sizable assemblies from sets of low-coverage sequence reads.

A. Comparative Assembly

As the number of sequenced organisms increases, alternative approaches to genome assembly based on orthologous relationships become ever more viable. Comparative *de novo* assembly algorithms map sequence reads to a high-quality reference genome and use the resultant anchoring information to direct the assembly process. Originally proposed by Pop [17], the AMOS comparative assembler employs an *alignment-layout-consensus* approach to genome assembly. AMOS uses a complete, high-quality sequence of a closely related organism to determine the placement of reads in a layout graph.

More recently, the related, but distinct concept of assisted assembly was proposed by Gnerre [18]. Designed for use with low-coverage sequences, assisted assembly reinforces information already present in reads to detect erroneous or missed overlaps during the initial phase of genome assembly. Simultaneously constructing both a *de novo* and a comparative assembly, proximity relationships between reads are used to guide the assembly process.

B. Hash Tables and Variability

Given the recent trend towards *k*-mer centric genome assembly, the application of a similar approach to comparative assembly is worthy of consideration. Although the use of *k*-mers has a long history in both sequence alignment [19, 20] and sequence assembly [10-12], the underlying implementation typically manifests itself in the form of hash tables or hash maps. Hash tables and maps are dictionary data structures that use a key and hashing function to provide rapid, **O**(1), access to a set of mapped values [21]. As hash maps are capable of quickly detecting exact matches between keys, they are an ideal data structure for use in *k*-mer centric alignment and assembly applications. In a hash data structure, the hash key is used to functionally determine a mapped value. The implication of this property is that, although redundancy is permitted among the values in a hash map, the hash keys must be unique. While this uniqueness requirement provides hash structures with the underlying property to facilitate speed, it constrains access to exact matches of keys. This renders hash data structures intolerant of variations in sequence composition, such as sequence errors, polymorphisms, insertions and deletions, common in biological sequences.

 ISBN: 978-1-61208-137-3

To circumvent the constraints imposed by the uniqueness requirement of hash keys, alignment applications have employed a number of different strategies. Chief among these is the "seed and extend" strategy used by BLAST [19], which applies a hash table to seed exact *k*-mer matches, before attempting to join high-scoring alignments using dynamic programming. An alternative approach is the use of spaced seeds [22], which permit a degree of mismatch at pre-determined positions in a sequence. However, both approaches facilitate sensitivity by sacrificing the speed inherent in the hash structure.

Richer, object-oriented, programming languages permit the extension of hash maps to provide built-in support for key variability. Originally proposed by Topac [23], a Fuzzy Hash Map (FHM) applies fuzzy capabilities to traditional hash structures, with a minimal reduction in access speed. FHMs permit a degree of variation in hash keys and can be used for inexact sequence comparison.

To illustrate the applicability of FHM to sequence alignment and assembly, a draft genome of *M.genitalium*, sampled at very low levels of coverage, was anchored against the complete genome of *M.pneumoniae*, which was then used to guide the assembly process. The approach is highly effective for both ordering and orientating low coverage sets of reads into assembly contigs and scaffolds. The remainder of this discussion includes a description of how fuzzy *k*-mers can be implemented using a FHM. This is followed by a description of the anchoring and assembly process and the presentation of results. Finally, the mechanism used to test the validity of the approach is described and conclusions presented.

II. FUZZY HASH MAPS AND FUZZY *K*-MERS

Unlike procedural programming languages, object-oriented languages allow arbitrary objects to act as keys and values in a hash map [21]. The rapid access time of hash maps is accomplished by transforming a key value to an integer value that corresponds to a table index. When a collision between a search term and a hash key is detected, this transformation is applied to provide access to the mapped value. In the Java language, the semantics of object equality is determined by the implementation of the *hashCode()* and *equals()* methods [24]. When searching a hash map for a given key, if two *hashCode()* methods return the same integer value, an initial collision is detected. The *equals()* method is then executed to resolve any ambiguity and determine if a full collision has occurred.

FHMs manipulate the relationship between both of these methods by encouraging initial collisions based on part of the hash key and using the *equals()* method to permit a degree of variability in the remainder of the key. The degree of similarity is determined by the implementation of the *equals()* method, which can employ any sequence similarity algorithm that is capable of returning a fuzzy value between 0 and 1.

As depicted in Figure 1, fuzzy *k*-mers can be accommodated in a FHM by specifying the part of the *k*-mer to be used when computing the hash code. The remainder of the *k*-mer is evaluated by encapsulating a sequence similarity algorithm inside the *equals()* method. Using standard object-oriented techniques such as composition and inheritance, any edit distance algorithm such as Levenshtein Distance [25], Hamming Distance [26] and the Smith-Waterman algorithm [27] can be used. Consistent with traditional "seed and extend" strategies, the design of a fuzzy hash key is a trade-off between speed and sensitivity. Computing a hash code on too small a part of a hash key has the effect of flattening the FHM into a list, reducing the speed in proportion to the time complexity of the sequence similarity algorithm. In practice, specifying a minimum word size of 11 bases in the *hashCode()* implementation allows variability in the remainder of a *k*-mer, with little or no impact on running time.

III. ANCHORING AND ASSEMBLING FUZZY K-MERS

To illustrate the relevance and utility of FHMs to comparative genome assembly, the approach was used to anchor and assemble a draft genome of *Mycoplasma genitalium* using the complete genome of *Mycoplasma pneumoniae* to direct part the assembly. A *k*-mer centric anchoring and assembly strategy was applied, which consists of four main phases: anchor detection and extraction, anchor alignment, contig assembly and contig scaffolding.

A. Anchor Detection and Extraction

Given a complete, high-quality reference genome, a de

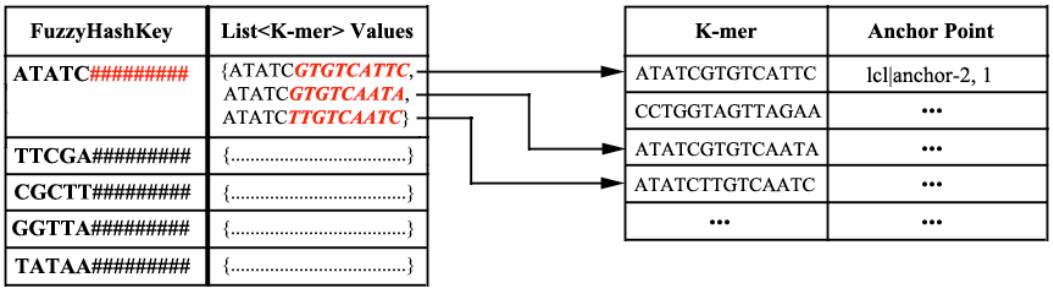

Figure 1. A Fuzzy Hash Map with a hash key initialised to cause collisions if the first five bases in a sequence are the same.

Bruijn graph can be created that represents a perfect tiling path through the genome. For a genome of size n, the de Bruijn graph will have $\mathbf{O}(n)$ nodes and $\mathbf{O}(n)$ edges, regardless of the number of reads in an assembly [28]. Each node in the graph represents a k-mer and can be weighted to reflect the multiplicity of matches to the sequence it contains. In the FHM approach, the multiplicity is not denoted by an integer value, but by labelling each node with read edges. A read edge represents the alignment of part of a genome with the k-mer contained by a node. Thus, nodes composed with more than one read edge represent repetitive sequences and are easily identified and, if necessary, avoided. The anchor detection and extraction process requires that the full reference genome be parsed and transformed into a de Bruijn graph (Figure 2). Although the memory requirements of a de Bruijn graph are huge, the memory consumption can be greatly reduced by merging all nodes with an in-degree and out-degree of 1. In the case of anchor detection, an additional constraint of merging only nodes with a multiplicity of 1 yields a sequence graph, where each merged edge represents a unique anchoring region. These anchoring regions are easily detected using a Depth-First Search [29] and are extracted and written to a FASTA file and to a serialized map. It is noteworthy that this process is executed once for each reference genome and is not undertaken as part of the assembly.

B. *Anchor Alignment*

Anchoring the reads from a draft genome requires that all reads, in both forward and reverse orientations, be compared against each anchor sequence. Before the alignment and assembly phases commence, the anchoring sequences are first parsed and read into a FHM. The FHM must be configured with a *FuzzyHashKey* that specifies the parts of each k-mer to be used to compute similarity. In addition, the *FuzzyHashKey* must also be configured with the sequence similarity algorithm to use and a fuzzy threshold value. Only alignment matches above the fuzzy threshold will cause a full collision in the FHM and indicate a match. Thus, a fuzzy threshold of 0.65 will only result in a match if a hash code causes an initial collision in the FHM and 65% of the remainder of the k-mer matches a hash key. Read alignment is accomplished by decomposing each read into a set of overlapping k-mers and attempting to add each k-mer to the FHM. If a match is found in the FHM, the name and index of the anchor is recorded, along with the orientation of the read. After the read has been aligned in both orientations, a majority count is use to determine the correct orientation of the read and its order with respect to the anchor. Each anchor maintains a list of the name, orientation and starting

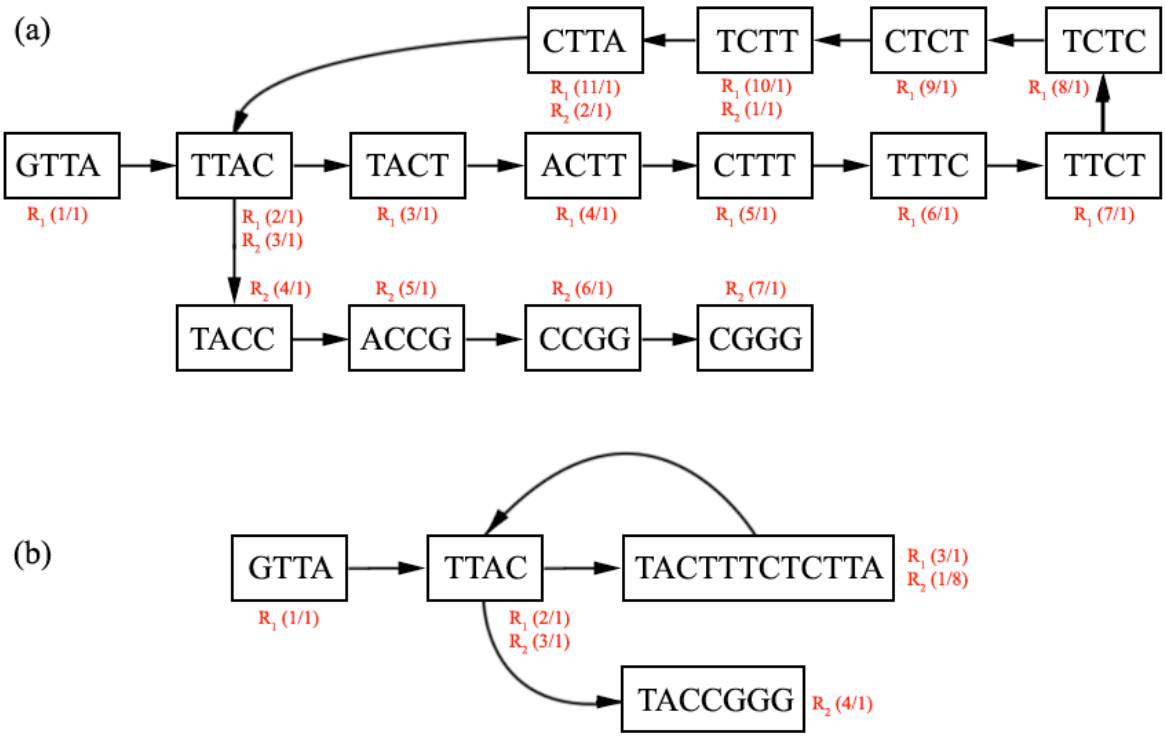

Figure 2. (a) A 4-mer de Bruijn graph for the overlapping sequences GTTACTTTCTCTTA and TCTTACCGGG. In practice k-mer sizes of at least 24 are used. As each read is added to the graph, the index position of the read (in red) relative to the sequence of the graph node is recorded. This information enables reliable transversal through highly repetitive nodes, by following the indices of the current read in increasing order. (b) Transformation to a sequence graph can be achieved by merging together nodes with an in-degree and out-degree of 1. The starting index of each read with respect to the merged node is altered to reflect the length of the newly merged sequence. The transformation to a sequence graph has the effect of significantly reducing the number of nodes and edges in the graph.

 ISBN: 978-1-61208-137-3

position of each read and automatically sorts the set of reads using a priority queue. As each anchor knows its own starting index with respect to the reference genome, the alignment process not only orientates, but also orders, each anchored read.

C. Contig Assembly

The assembly of overlapping reads into contigs is based on the application of a de Bruijn graph, in a manner similar to that used for anchor detection and extraction. As each draft read is parsed, an attempt is made to anchor the read using the procedure described above. If a read has been anchored, it is added to the de Bruijn graph only in the orientation given by the anchor. Otherwise, a read is added to the graph in both orientations. After parsing the full set of draft reads, the de Bruijn graph is transformed into a sequence graph by merging together all nodes with an in-degree and out-degree of 1. Assembly commences by generating a stack representing the set of source nodes in the graph. At low levels of coverage, there is an insufficient number of overlapping reads to provide a single path through the graph. Thus, the total number of source nodes in the graph indicates the minimum number of contigs available to the assembler.

Contig assembly is facilitated by the labelling of graph nodes with read edges. Starting at a source node, the list of read edges is processed to identify the starting sequence of an unprocessed read. The best candidate read edge can be identified by its index with respect to its own read and its index relative to the graph node. Using the current read as a heuristic value, neighbouring edges are evaluated and offered to a priority queue. The priority queue applies a distance constraint to determine if a read edge is permissible and, after examining all neighbouring edges, returns the next best edge to the assembler. By applying a distance constraint in this manner, the assembler will select the correct path when it encounters a branch in the graph. In addition to selecting the next edge, the priority queue is also responsible for determining the next read to process.

The labelling of nodes with read edges also facilitates the transversal of loops in the graph, which represent repetitive sequences. Using the current read as a heuristic, if an adjacent node has more than one read edge for the current read, the priority queue will only select a read edge that meets the distance constraints.

D. Contig Scaffolding

Given a set of contigs from the initial assembly phase, the anchoring information can now be applied to order and group the contigs. Although the absolute index of each anchor sequence with respect to the reference genome is known, this information cannot be used to determine the distance between contigs, as there may be large insertions or deletions in the reference sequence. Thus, the scaffolding of contigs involves the full ordering of contigs and the grouping of contigs linked by anchors into sub-contigs.

This final phase of assembly involves polling each anchor from a sorted queue of anchors. If an anchor spans more than one contig, at least two reads must be present in the adjacent contig to establish a join. As each anchor is processed, its aligned reads are removed in ascending order of alignment index. When all the reads have been polled from an anchor, the next anchor is removed from the anchor queue and the process iterates. The final assembly output is a FASTA file containing the assembled reads and an XML document. The XML document contains information about the set of contigs, including the constituent reads, read index and orientation.

IV. RESULTS AND DISCUSSION

The 0.58Mb genome of *M.genitalium* was randomly sampled at coverage levels from 0.1X-2.0X and assembled using the 0.81Mb genome of *M.pneumoniae* as a reference sequence. Using a *k*-mer size of 24 and, with fuzzy index and fuzzy threshold values of 11 and 0.8 respectively, the FHM approach anchored 65.56% of the *M.genitalium* reads. Empirical evidence demonstrates that, for genomic sequences, as the fuzzy index decreases below 11, the number of collisions in the FHM increase exponentially until the access time reaches $\mathbf{O}(n)$, at which point the running time of the FHM is no better that that of an indexed list. The fuzzy threshold of 0.8 reflects the close genetic relationship between *M.genitalium* and *M.pneumoniae*. For more divergent species, this parameter should be relaxed to permit a greater tolerance of sequence variability during the anchoring phase.

The results of the assembly at various levels of coverage are shown in Table 1. Among the more salient features of the fuzzy assembly approach, is the low percentage of orientation errors. This illustrates the utility of genome anchoring in general and the FHM in particular, for determining the correct orientation of a sequence read, even at very low levels of coverage.

TABLE I. SUMMARY OF ASSEMBLY RESULTS AT VARYING LEVELS OF COVERAGE.

Coverage	N50 Contig	N50 Scaffold	% Ordering Errors	% Orientation Errors	Time (s)
2.0	2141	51215	1.21	0.12	19.2
1.8	1918	14787	4.60	1.00	17.3
1.6	1798	14853	1.12	0.17	16.2
1.4	1739	9734	0.39	0.59	14.6
1.2	1456	10322	2.18	0.46	12.8
1.0	1269	6001	1.24	0.97	11.1
0.8	1228	8240	1.55	0.69	7.5
0.6	992	4743	0.92	0.46	7.8
0.4	-	2450	2.07	0.00	6.0
0.2	-	2539	1.38	2.07	4.1

The N50 metric indicates that 50% of bases are in contigs of size *n* or greater. Again, the effectiveness of the approach can be seen by comparing the N50 size for the contigs generated by the initial assembly with the N50 size of scaffolded contigs. Even at ultra-low levels of coverage,

 ISBN: 978-1-61208-137-3

the assembler is capable of generating sizable contigs with low ordering and orientation errors. The execution time exhibits a logarithmic growth rate in the order **O**(log *n*). Allowing for the parsing of an ever-increasing number of reads, this slow growth rate illustrates that using fuzzy *k*-mers in this manner does not compromise running time.

V. VALIDATION OF APPROACH

To establish the validity of the approach, an automated testing framework was developed that is capable of examining and scoring the order and orientation of each read in the assembly. The set of reads for each draft genome was randomly sampled from a complete genome, by providing information such as desired coverage level, average read length and clone insert length to a validation framework. The output of the read generation process is a set of randomly sampled and oriented reads in FASTA format, representing the draft genome, and a specialised data structure containing validation data for each read. The validation data includes the correct order and orientation of reads, the length of each read and the distance between adjacent reads.

After anchoring and assembling the draft genome, the set of assembled reads was validated, by computing a local alignment of each contiguous set of reads against the full ordered set of randomly sampled sequences. This was accomplished by creating a dynamic programming matrix and scoring the list of reads in each contig against the full list of reads generated by the framework. It should be noted that the dynamic programming matrix requires only the names of reads and their relative distances to compute an alignment score. Furthermore, a positive score in the programming matrix requires a read to be both in order and at the correct distance relative to its adjacent reads.

In addition to ascertaining the correct order of reads, the test framework also computes, from a suffix of the FASTA sequence name, the correct orientation of each read. The local alignment was implemented using a modification of the Smith-Waterman [27] algorithm. Ancillary information, such as N50 size and Lander-Waterman [14] statistics, is also generated by the validation framework.

VI. CONCLUSION

The anchoring and assembly process described is capable of rapidly ordering and orientating reads from a draft genome with a low level of errors. In particular, the anchoring mechanism is highly effective in orientating reads, thereby reducing the size of the graph created by the assembler and simplifying the assembly of contigs. Directing assembly using sets of anchored reads enables the construction of large contig scaffolds, even at low levels of coverage. Furthermore, the application of a FHM data structure permits a degree of variability between sequences, without sacrificing execution speed. Finally, the fuzzy k-mer approach allows a high degree of error tolerance that is invaluable when processing biological sequences that contain sequence errors, insertions and deletions.

VII. REFERENCES

[1] S. Batzoglou, "The many faces of sequence alignment," *Briefings in bioinformatics,* vol. 6, p. 6, 2005.

[2] H. Li and N. Homer, "A survey of sequence alignment algorithms for next-generation sequencing," *Brief Bioinform,* p. bbq015, 2010.

[3] M. Schatz, A. Delcher, and S. Salzberg, "Assembly of large genomes using second-generation sequencing," *Genome Research,* vol. 20, p. 1165, 2010.

[4] P. Flicek and E. Birney, "Sense from sequence reads: methods for alignment and assembly," *Nature Methods,* vol. 6, pp. S6-S12, 2009.

[5] C. Fuller, L. Middendorf, S. Benner, G. Church, T. Harris, X. Huang, S. Jovanovich, J. Nelson, J. Schloss, and D. Schwartz, "The challenges of sequencing by synthesis," *nature biotechnology,* vol. 27, pp. 1013-1023, 2009.

[6] C. Hutchison III, "DNA sequencing: bench to bedside and beyond," *Nucleic Acids Research,* 2007.

[7] J. Shendure and H. Ji, "Next-generation DNA sequencing," *nature biotechnology,* vol. 26, pp. 1135-1145, 2008.

[8] A. Sundquist, M. Ronaghi, H. Tang, P. Pevzner, and S. Batzoglou, "Whole-genome sequencing and assembly with high-throughput, short-read technologies," *PLoS One,* vol. 2, 2007.

[9] J. Kececioglu and E. Myers, "Combinatorial algorithms for DNA sequence assembly," *Algorithmica,* vol. 13, pp. 7-51, 1995.

[10] J. Butler, I. MacCallum, M. Kleber, I. Shlyakhter, M. Belmonte, E. Lander, C. Nusbaum, and D. Jaffe, "ALLPATHS: De novo assembly of whole-genome shotgun microreads," *Genome Research,* vol. 18, p. 810, 2008.

[11] J. Simpson, K. Wong, S. Jackman, J. Schein, S. Jones, and Birol, "ABySS: A parallel assembler for short read sequence data," *Genome Research,* vol. 19, p. 1117, 2009.

[12] D. Zerbino and E. Birney, "Velvet: Algorithms for de novo short read assembly using de Bruijn graphs," *Genome Research,* vol. 18, p. 821, 2008.

[13] R. Fleischmann, M. Adams, O. White, R. Clayton, E. Kirkness, A. Kerlavage, C. Bult, J. Tomb, B. Dougherty, and J. Merrick, "Whole-genome random sequencing and assembly of Haemophilus influenzae Rd," *Science,* vol. 269, p. 496, 1995.

[14] E. Lander and M. Waterman, "Genomic mapping by fingerprinting random clones: a mathematical analysis," *Genomics,* vol. 2, pp. 231-239, 1988.

[15] I.-M. A. C. Konstantinos Liolios, Konstantinos Mavromatis, Nektarios Tavernarakis, Philip Hugenholtz, Victor M. Markowitz and Nikos C. Kyrpides, "The Genomes On Line Database (GOLD) in 2009: status of genomic and metagenomic projects and their associated metadata," *Nucleic Acids Research,* vol. 38, 2010.

[16] N. Hall, "Advanced sequencing technologies and their wider impact in microbiology," *Journal of Experimental Biology,* vol. 210, p. 1518, 2007.

[17] M. Pop, A. Phillippy, A. Delcher, and S. Salzberg, "Comparative genome assembly," *Briefings in bioinformatics,* vol. 5, p. 237, 2004.

[18] S. Gnerre, E. Lander, K. Lindblad-Toh, and D. Jaffe, "Assisted assembly: how to improve a de novo genome assembly by using related species," *Genome biology,* vol. 10, p. R88, 2009.

[19] S. Altschul, W. Gish, W. Miller, E. Myers, and D. Lipman, "Basic local alignment search tool," *Journal of Molecular Biology,* vol. 215, pp. 403-410, 1990.

[20] W. Pearson and D. Lipman, "Improved tools for biological sequence comparison," *Proceedings of the National Academy of Sciences,* vol. 85, p. 2444, 1988.

[21] M. Goodrich and R. Tamassia, "Data Structures and Algorithms in Java," John Wiley & Sons, 2001.

[22] B. Ma, J. Tromp, and M. Li, "PatternHunter: faster and more sensitive homology search," *Bioinformatics,* vol. 18, p. 440, 2002.

 ISBN: 978-1-61208-137-3

[23] V. Topac, "Efficient fuzzy search enabled hash map," 2010, pp. 39-44.

[24] J. Gosling, B. Joy, G. Steele, and G. Bracha, *Java (TM) Language Specification, The (Java (Addison-Wesley))*: Addison-Wesley Professional, 2005.

[25] V. Levenshtein, "Binary codes capable of correcting deletions, insertions, and reversals," 1966.

[26] R. Hamming, "Error detecting and error correcting codes," *Bell System Technical Journal,* vol. 29, pp. 147-160, 1950.

[27] T. Smith and M. Waterman, "Identification of common molecular subsequences," *Journal of Molecular Biology,* vol. 147, pp. 195-197, 1981.

[28] M. Chaisson and P. Pevzner, "Short read fragment assembly of bacterial genomes," *Genome Research,* vol. 18, p. 324, 2008.

[29] R. Tarjan, "Depth-first search and linear graph algorithms," 1971, pp. 114-121.

 ISBN: 978-1-61208-137-3