A Fast Sequence Assembly Method Based on Compressed Data Structures

Peifeng Liang, Yancong Zhang, Kui Lin and Jinglu Hu

Abstract— **Assembling a large genome using next generation sequencing reads requires large computer memory and a long execution time. To reduce these requirements, a memory and time efficient assembler is presented from applying FM-index in JR-Assembler, called FMJ-Assembler, where FM stand for FM_R-index derived from the FM-index and BWT and J for jumping extension. The FMJ-Assembler uses expanded FM-index and BWT to compress data of reads to save memory and jumping extension method make it faster in CPU time. An extensive comparison of the FMJ-Assembler with current assemblers shows that the FMJ-Assembler achieves a better or comparable overall assembly quality and requires lower memory use and less CPU time. All these advantages of the FMJ-Assembler indicate that the FMJ-Assembler will be an efficient assembly method in next generation sequencing technology.**

I. INTRODUCTION

Since the first DNA sequences were obtained in the early 1970s, the advent of DNA sequencing has significantly accelerated biological and biomedical research and discovery. Several sequencing strategies such as the chain termination method and shotgun methods were developed for sequencing from short DNA fragments to even the full genome. Over the past several years, next-generation DNA sequencing technologies have catapulted to prominence with increasingly widespread adoption of several platforms that individually implement different flavors of massively parallel cyclic-array sequencing. At the same time, the efficient computational techniques are required to process and analyze the data. High throughput sequencing has also generated new analysis challenges, it needs more memory and CPU time. As whole genome sequencing is now a routine experimental measure, efficient algorithms and softwares in saving memory or CPU time are needed that can scale to match the data generated. This is particularly important for the computationally demanding de novo assembly problem.

Many assembly algorithms for the next-generation sequencing technologies based on eulerian path and de Bruijn graph, such as Velvet [10] and ALLPATHS [4] require much memory space, for they need much graph calculation. The other techniques used for reconstructing the underlying sequence from the short fragments are how to save memory in assembly algorithms. Recently, some techniques were introduced for reducing memory consumption, which included the use of sparse graph representations, compressed graph data structures, Bloom filters and the FM-index for efficient overlap calculation [6]. This new class of memory efficient assemblers allows the analysis of much larger data sets. But in practice, most of this kind of algorithms take too much CPU time.

JR-Assembler [2] is an efficient algorithms in saving CPU time. Because it is based on overlap-layout, it needs less memory space than the de Bruijn graph based approaches. The JR-Assembler builds index based on binary search tree rather than compressed data structure, however, it takes more memory space than some compressed data structure based approaches, such as SGR-Assembler [8]. FM-index [3] is an efficient method in searching a substring in a compressed string and saving memory space. Here, we expand FM-index of a single string to a set of reads' FM-index, and apply it in JR-Assembler to design a new read assembly method, called FMJ-Assembler. The proposed FMJ-Assembler applies FM-index because it is an efficacious structure in saving memory. And jumping extension method can speed up extension and readily jump over small repeats. For large genomes, the FMJ-Assembler is an efficient in memory use and run time.

II. THE EXPANDED FM-INDEX

The advantage of the FMJ-Assembler is that it uses a expanded FM-index structure to build a searching index for a set of reads, which make the FMJ-Assembler use less memory space and search overlap between the reads rapidly. In this section, the expanded FM-index of a set of reads is introduced called FM_R-index [8], which is expanded from suffix array(SA), BWT and FM-index [3]. The FM_R-index is used to build index for reads in the in the FMJ-Assembler. Using FM_R-index of the reads data, the FMJ-Assembler can search overlaps between reads rapidly using *backwardsearch* method [3] in seed extension stage.

Since the FMJ-Assembler processes a collection of reads, the data structures of SA, BWT and FM-index can't be used in it. The SA can be expanded a suffix array of a set of strings. Consider a set of strings $T = \{T_1, T_2, \ldots, T_m\}$ over a constant-size alphabet Σ. We assume that the total length of all strings collections as n and each text T_i starts with a special character \$ in Σ, where \$ is alphabetically smaller than all other characters in Σ and it does not appear in any other part of a text. For i in $[1,n]$, define:

$$SA_c[i] = (j,k) \tag{1}$$

if $T_j[k,|T_j|]$ is the i-th lowest suffix in T. And the definition

Peifeng Liang and Jinglu Hu are with the Graduate School of Information Production and Systems, WASEDA University, 2-7 Hibikino, Wakamatsu-ku, Kitakyushu, Fukuoka, Japan, (email: liangpeifeng@akane.waseda.jp;jinglu@waseda.jp).

Yancong Zhang and Kui Lin are with College of Life Sciences, Beijing Normal University, 19 Xinjiekou Outer St, Haidian, Beijing, China, (email: linkui@bnu.edu.cn; zhangyc201211@gmail.com).

978-1-4244-7929-0/14/$26.00 ©2014 IEEE

of BWT is expanded to a set of strings as follow:

$$BWT_c[i] = \begin{cases} T_j[k-1] & k > 1 \\ \$ & k = 1 \end{cases} \quad (2)$$

Like the BWT of a single string, BWT_c is a permutation of the symbols in T. The auxiliary data structures for the FM_R-index $C_c(a)$ and $Occ_c(a,i)$ are the same as $C(a)$ and $Occ(a,i)$ in [3].

III. ALGORITHM OVERVIEW

In this section, the proposed FMJ-Assembler's brief overview is introduced. The method of the FMJ-Assembler is similar as the JR-Assembler, which is based on jumping extension. Specifically, this method includes the following parts: preprocessing of raw reads, building FM_R-index for preprocessed reads, assembling reads and detecting repeats. Preprocessing of reads includes correcting bases error in raw reads and other necessary procedures that can improve assembly quality. Assembling reads applies assembly technique to merge reads onto longer fragments. In detecting repeats stage, assembler finds repeats and processes them, then outputs contigs. Fig.1 depicts the flow of data through the algorithm pipeline.

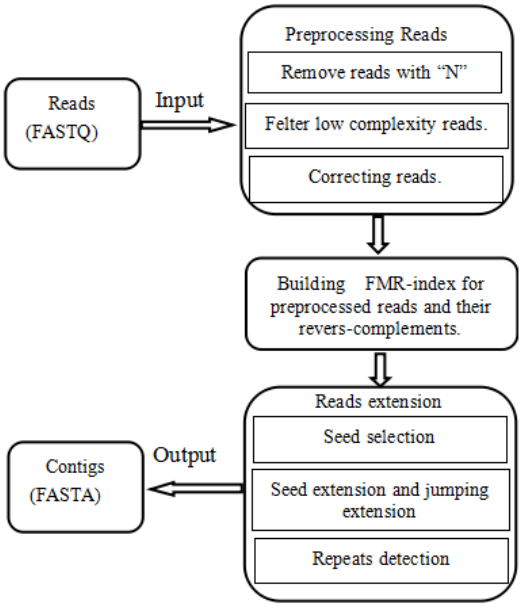

Fig. 1. The diagram of the flow of algorithm

A. Preprocessing raw reads

Preprocessing raw reads which is very important for a assembler to get high quality assembly result includes filtering out bad reads and correcting bases' errors based k-mer frequencies. In the FMJ-Assembler, bad reads that contain any "N" or any low complexity region are filtered out from raw dataset, as bad reads containing "N" is noninformative or bad reads containing any low complexity region may lead to false positive overlaps with other reads. When input reads have too many sequencing errors at the 3'end, assembler algorithm may not be able to find correct overlapping reads for selected seeds to do the extension and may thus generate a set of very short contigs. So the FMJ-Assembler trims input reads at 3'end into shorter but better quality reads. Then the FMJ-Assembler corrects trimmed reads using a correction algorithm based k-mer [5], which is an efficient correction algorithm with less memory.

B. Building FM_R-index and Overlap detection

1) Building FM_R-index: After Processing and Correcting raw reads, the FMJ-Assembler collapses them into one unique read and records its frequency c. Then the FMJ-Assembler computes SA_c and FM_R-index for identical reads and their reverse-complements.

Considering R be a set of preprocessed reads. To build the FM_R-index of R, we must first compute the generalized SA_c of R. Let $S = R_1, R_2, \ldots, R_m$ be a concatenation of all members of R. SA_c can be computed using a efficient algorithm [7]. Once SA_c has been constructed, the BWT_c of R, and hence the FM_R-index is easily computed as described in section II. We also compute the FM_R-index for the set of reversed reads, denoted R^r, which is necessary to compute overlaps between reverse complemented reads. We also output the lexographic index of R, which is a permutation of the indices $\{1,2,\ldots,|R|\}$ of R sorted by the lexographic order of the strings. This can be found directly from SA_c and is used to determine the identities of the reads in R from the suffix array interval positions once an overlap has been found.

2) Overlap detection based on FM_R-index: Before introducing read extension, we first present the method of overlap detection based on FM_R-index, which will be used in next subsection. It is important to detect overlap between reads in this algorithm, as the FMJ-Assembler is a overlap-layout based algorithm. The FMJ-Assembler needs to search a read's overlap with other reads rapidly. This procedure is similar to alignment method which searchs the position of a substring in reference string. In [3], *backwardsSearch* method searches for a string P in T using $C(a)$ and $Occ(a,1)$. Let S is a string whose suffix array interval is known to be $[l,u]$. The interval for the string aS can be calculated from $[l,u]$ using C and Occ by the following:

$$l' = C(a) + Occ(a, l-1) \quad (3)$$

$$u' = C(a) + Occ(a, u) - 1 \quad (4)$$

After constructing SA_c and BWT_c, the set of t_{min} overlaps between reads in R can be computed. Let X be an arbitrary read in R. After performing k steps the *backwardsSearch* procedure [3] on the string X, the interval $[l,u]$ for the suffix of length k of X can be calculated. The reads indicated by the suffix array entries in $[l,u]$ therefore have a substring that matches a suffix of X, denoted Q. Then the interval for the strings beginning with Q can be determined by calculating the interval for the string Q using equations (3) and (4). The algorithm is presented below in $findOverlaps$.

After preprocessing the raw reads and building FM_R-index for preprocessed reads, the FMJ-Assembler begins assembly reads with jumping-extension method. This procedure similar to [2], that includes selecting good seed, extending seed and processing repeat. After this stage, contigs would be got.

Algorithm 1 findOverlaps

Input: Read X; minimal length overlap t.

Output: the interval $[l,u]$ for reads that have a prefix matching the suffix of X;

```
1: function FINDOVERLAP(X, t)
2:     i ← |X|
3:     l ← C_c(X[i])
4:     u ← C_c(X[i]+1) − 1
5:     i ← i − 1
6:     while (l ≤ u)&(i ≥ 1) do
7:         if |X| − i + 1 ≥ t then
8:             [l_$, u_$] ← updateBackwards([l,u], $)
9:             if l_$ ≤ u_$ then
10:                outputOverlaps (X, [l_$, u_$])
11:            end if
12:        end if
13:        [l,u] ← updateBackward ([l,u], X[i])
14:        i ← i − 1
15:    end while
16:    if l ≤ u then
17:        outputContained (X, [l,u])
18:    end if
19: end function
```

C. Read extension

1) Seed Selection: In an extension-based assembler, a good seed should not contain any sequencing errors and should not be selected from a repeat region. A read containing sequencing errors usually has a very low read count. On the other hand, a read from a repeat region usually has a high read count because identical reads from other repeat loci are counted as well. Thus, in seed selection, reads with a very low or a very high read count should be avoided. Those read counts ranked in between 1% and 25% are selected as the seeds for extension.

2) Seed Extension: In this stage, the FMJ-Assembler extends seeds' 3'end and 5'end into a long fragment. Given a seed, the FMJ-Assembler first extends its 3'end and then its 5'end. To extend a seed R_{seed} at the 3'end, the FMJ-Assembler searches all unassembled unique reads for extendable reads. A read is extendable for R_{seed} if its 5'end overlaps with the 3'end of R_{seed} and its 3'end overlaps with one or more unique reads, for example, $R_1, R_2, \cdots, R_5$ in Fig.2.

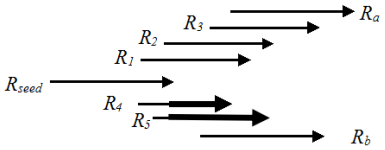

Fig. 2. The diagram of two extension candidates

In Fig.2 two extension candidates, R_a and R_b, are found for a jumping extension. The bold parts in R_4, R_5 indicate that the corresponding bases are identical between R_4 and R_5 but cannot be aligned with the corresponding regions of R_1, R_2 and R_3. In case 1, three connecting reads R_1, R_2 and R_3 connect R_{seed} and R_a. This process is shown as Fig.3.

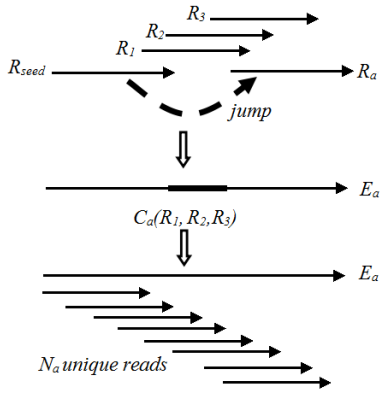

Fig. 3. The diagram of jumping extension A

The process of connecting R_{seed} and a unique read is termed a jump in this study. In Fig.3, to fill the gap between R_{seed} and R_a, the FMJ-Assembler constructs a consensus sequence $C_a(R_1,R_2,R_3)$ from the connecting reads, shown in bold line. After a jump, an extension candidate E_a is built by concatenating R_{seed}, $C_a(R_1,R_2,R_3)$ and R_a. Similarly, in case 2 another extension candidate E_b is built by concatenating R_{seed}, $C_a(R_4,R_5)$ and R_b.

When a R_{seed} comes from a repeat, there can be more than one extension candidate. The FMJ-Assembler has to decide which one is better. Define a supporting score S_a:

$$S_a = \frac{N_a}{l_a - l + 1} \tag{5}$$

where l_a is the length of E_a, l is the length of reads and N_a is the number of unique reads in R that can be remapped onto E_a(Fig.3). If the score S_a is less than the cutoff (0.3 by default), that is, without enough reads support the extension, which is neglected. In this algorithm, the FMJ-Assembler selects the candidate with the highest score for extension and deals with the repeat problem later. All unique reads that can be remapped onto this extension are then labeled as assembled. Assembled unique reads will not be used in the remaining extensions. The process is repeated until the 3' and 5'ends cannot be extended anymore.

In some condition, an extension may not be proceeded because of sequencing errors at the read tail. The FMJ-Assembler uses a back trimming method to solve this problem. The FMJ-Assembler trims one base from the end of the last extended sequence and checks whether a jump is possible from the trimmed sequence. If not, the trimming continues until a jump is possible or until the previous extension is reached. When a jump is made, the extension and back trimming procedures resume until no jump can be made further. Once the extension at the 3'end is done, the FMJ-Assembler starts to extend the 5'end of the initial seed by the same two procedures. After the extensions at both ends are completed, a long fragment is obtained. After completing extensions, the FMJ-Assembler has to detect repeat using a repeat detection method [2]. After this procedure, contigs can be obtained. In this algorithm, merging contigs to longer

scaffolds is not considered, but SSPACE [1] or SOAPdenovo [5] can be used to scaffold the contigs.

IV. Result and Discussion

A. Datasets and Environments

In order to evaluate the effectiveness of this algorithm, we employ two SRS datasets of E.coli and S.roseosporus from the National Center for Biotechnology Information (NCBI) Short Read Archive (SRA) with the accession nos SRX016044 and SRX026747. We compared the FMJ-Assembler with other algorithm: JR-Assembler [2], SGA [8], ABySS1.2.6 [9], Velvet1.0.19 [10], SOAPdenovo2 [5] and ALLPATHS-LG [4]. These algorithms are implemented in Linux operating system, and the computational experiments are carried out on a Dell Server with a 1.6GHz eight-core Intel Xeon E5310 processors and 48GB of RAM.

B. Comparison for Assembling the Escherichia coli Genome

E.coli genome provides a good real-world test case for assembly algorithms because it has a complete and accurate reference sequence. The dataset consists of 10.3M read pairs sequenced using the Illumina Genome Analyzer II. The mean DNA fragments size is 300 bp from which reads of length 100 bp were taken from both ends of the fragments.

As sequence assemblers are often sensitive to the input parameters. The de Bruijn graph assemblers were run for all odd k-mer sizes between 51 and 73 (inclusive). The k-mer size was selected for further analysis (57 for ABySS, 53 for Velvet, 59 for SOAPdenovo and ALLPATHS-LG, 41 for SGA error correction, 55 for SGA's minimum overlap). Similarly, for JR-Assembler and the FMJ-Assembler, we set 30 for minimum overlap and 45 for maximum overlap. In FMJ-Assembler, we trim read's 3'end to reduce base errors, and length of reads become 85 bps.

TABLE I
ASSEMBLY STATISTICS OF THE E.COLI DATASET BY DIFFERENT ASSEMBLERS

	contigs	Sum*	Max	Mean	N50	time$	Mem&
FMJ	211	4.53	178,523	21,469	42,869	4.7	5.5
JR	201	4.53	195,963	22,537	43,656	4.3	6.3
SOAP*	286	4.53	120,146	15,839	32,163	3.2	20.2
ABySS	245	4.54	140,105	18,530	37,656	12.3	28.4
SGA	436	4.53	135,365	10,389	20.357	18.1	5.0
velvet	220	4.54	140,114	20,636	42,556	8.0	30.2
ALL*	221	4.54	180,325	20,542	45,568	10.6	35.0

SOAP*: SOAPdenovo, ALL*: ALLPATHS-LG, *: Mbps, $: hour, &: GB.
*N50 is the size of the smallest contig such that 50% of the assembled bases are in the contigs of size equal to or larger than the N50 value.
*Contigs of length < 300 bp were not counted.

Table I shows the assembly statistics by these seven assemblers. The FMJ-assembler was better than the JR-assembler in saving memory and better than SGA in run time. But the FMJ-assembler used more memory than SGA. SGA used compression structure in all process, which was the reason of taking more CPU time. In the FMJ-Assembler, preprocessing reads did not work in compression structure. This process required more memory than SGA. Because building FM-index took some CPU time, the FMJ-Assembler spent more time than JR-assembler. For assembly result, the FMJ-Assembler was close to JR-assembler but better than other assembler in most assembly metrics. ABySS, SOAPdenovo and ALLPATHS-LG did not use compression structure, so they required more memory 20-35 GB.

C. Comparison for Assembling the S.roseosporus Genome

S.roseosporus genome included 7.7Mb mega base pairs. We assembled it with the FMJ-Assembler, JR-assembler and SOAPdenovo. ABySS and ALLPATHS-LG required too much memory to run completely and SGA took too much time. The results as Table II. From the result in this table, the FMJ-Assembler can save much memory and similar to general algorithm in run time. All these results demonstrate that the FMJ-Assemble could be an effective approach to sequence assemble in saving memory and run time.

TABLE II
ASSEMBLY STATISTICS OF THE S.ROSEOSPORUS

	contigs	Sum*	Max	Mean	N50	time$	Mem&
FMJ	1210	7.65	43,521	6,322	10,645	11.7	28
JR	1195	7.67	40,345	6,418	11,256	10.2	32
SOAP*	2256	7.65	30,146	3,391	4,703	6.2	39

SOAP*: SOAPdenovo, *: Mbps, $: hour, &: GB.

V. CONCLUSIONS

This work presents an assemble algorithm called FMJ-Assembler based on FM_R-index and Jumping extension. The most important feature of the FMJ-Assembler is that it can save memory while doesn't require too much CPU time. For Illumina reads, such as SRS data of genomes, FMJ-Assembler requires less memory than JR-Assembler and other algorithm based on de Bruijn graph and is more efficient than SGA-Assembler in CPU time.

References

[1] M. Boetzer, C. V. Henkel, H. J. Jansen, D. Butler, and W. Pirovano, "Scaffolding pre-assembled contigs using sspace," *Bioinformatics*, vol. 27, no. 4, pp. 578–579, 2011.

[2] T.-C. Chu, C.-H. Lu, T. Liu, G. C. Lee, W.-H. Li, and A. C.-C. Shih, "Assembler for de novo assembly of large genomes," *Proceedings of the National Academy of Sciences*, vol. 110, no. 36, pp. E3417–E3424, 2013.

[3] P. Ferragina and G. Manzini, "Opportunistic data structures with applications," in *Foundations of Computer Science, 2000. Proceedings. 41st Annual Symposium on*. IEEE, 2000, pp. 390–398.

[4] S. Gnerre, I. MacCallum, D. Przybylski, F. J. Ribeiro, J. N. Burton, B. J. Walker, T. Sharpe, G. Hall, T. P. Shea, S. Sykes *et al.*, "High-quality draft assemblies of mammalian genomes from massively parallel sequence data," *Proceedings of the National Academy of Sciences*, vol. 108, no. 4, pp. 1513–1518, 2011.

[5] R. Luo, B. Liu, Y. Xie, Z. Li, W. Huang, J. Yuan, G. He, Y. Chen, Q. Pan, Y. Liu *et al.*, "Soapdenovo2: an empirically improved memory-efficient short-read de novo assembler," *GigaScience*, vol. 1, no. 1, p. 18, 2012.

[6] N. Nagarajan and M. Pop, "Sequence assembly demystified," *Nature Reviews Genetics*, vol. 14, no. 3, pp. 157–167, 2013.

[7] S. J. Puglisi, W. F. Smyth, and A. H. Turpin, "A taxonomy of suffix array construction algorithms," *ACM Computing Surveys (CSUR)*, vol. 39, no. 2, p. 4, 2007.

[8] J. T. Simpson and R. Durbin, "Efficient de novo assembly of large genomes using compressed data structures," *Genome Research*, vol. 22, no. 3, pp. 549–556, 2012.

[9] J. T. Simpson, K. Wong, S. D. Jackman, J. E. Schein, S. J. Jones, and İ. Birol, "Abyss: a parallel assembler for short read sequence data," *Genome research*, vol. 19, no. 6, pp. 1117–1123, 2009.

[10] D. R. Zerbino and E. Birney, "Velvet: algorithms for de novo short read assembly using de bruijn graphs," *Genome research*, vol. 18, no. 5, pp. 821–829, 2008.