CS2A: a compressed suffix array-based method for short read alignment

Hongwei Huo*, Zhigang Sun*, Shuangjiang Li*, Jeffrey Scott Vitter†,
Xinkun Wang‡, Qiang Yu* and Jun Huan§

*School of Computer Science and Technology, Xidian University, China
†Department of Computer and Information Science, the University of Mississippi, USA
‡Department of Biochemistry and Molecular Genetics, Northwestern University, USA
§Department of Electrical Engineering and Computer Science, the University of Kansas, USA

Abstract

Next generation sequencing technologies generate enormous amount of short reads, which poses a significant computational challenge for short read alignment. Furthermore, because of sequence polymorphisms in a population, repetitive sequences, and sequencing errors, there still exist difficulties in correctly aligning all reads. We propose a space-efficient compressed suffix array-based method for short read alignment (CS2A) whose space achieves the high-order empirical entropy of the input string. Unlike BWA that uses two bits to represent a nucleotide, suitable for constant-sized alphabets, our encoding scheme can be applied to the string with any alphabet set. In addition, we present approximate pattern matching on compressed suffix array (CSA) for short read alignment. Our CS2A supports both mismatch and gapped alignments for single-end and paired-end reads mapping, being capable of efficiently aligning short sequencing reads to genome sequences. The experimental results show that CS2A can compete with the popular aligners in memory usage and mapping accuracy. The source code is available online.

I. Introduction

High-throughput next generation sequencing (NGS) technologies deliver millions of short reads in a single run. It makes the analysis of gene expression and genome variation on a genome-wide scale possible [19]. The first step in these bioinformatics analysis is short read alignment, which maps short reads to some positions on a reference genome (usually billions of base pairs), based upon the principle that the reads are approximate substrings of the reference [20]. These reads may have errors and the orientation of the read relative to the reference genome is usually not known, and we usually do not have the exact reference genome for these reads. How to align short reads to a reference genome, while account for the inexact pattern match, in the presence of sequencing errors, also within a reasonable amount of time and memory space, poses a significant computational challenge.

In recent years, many short read alignment algorithms have been developed. A classical short reads alignment framework used by earlier aligners is the seed and extension approach which extends alignment from a seed in various ways including backward searching and dynamic programming. MAQ [12] hashes short reads as seed and scans the reference genome to find candidate alignment locations. Then MAQ extends the remaining portion of the read against the reference to find an alignment which minimizes the sum of quality values of mismatched bases. Bowtie [10] uses Burrows-Wheeler Transform (BWT) [2]. BWA [11], another BWT-based aligner, adopts improved backward search to align short reads. There are also other methods [1], [15], [17] that use this framework.

Another short read alignment framework is based upon the use of spaced seeds. In this framework, several parts of a read are selected as spaced (or gapped) seeds. SOAP [16], for example, is based upon the seed (spaced seed)-and-extend approach. Using spaced seeds, aligners select more than one candidate alignment location (CAL). These candidate locations are then filtered by well-designed rules. At last, these candidate locations are verified by aligning remaining non-seed subsequences to the reference genome.

There are also aligners using different alignment frameworks [9], [18], [21]. Though there are many short read aligners available, the difficulties in correctly aligning all reads remain [14] because of sequence polymorphisms among individuals, repetitive sequences, and sequencing errors.

The compressed suffix array (CSA) introduced by Grossi and Vitter [4] is the first self-index that was proved by Grossi et al. [5] to achieve asymptotic space optimality in entropy sense. In this paper, we propose a space-efficient CSA-based method for short read alignment (CS2A). In CS2A, we construct the compressed index for the reference genome sequence based upon the storage scheme developed by Huo et al. [7]. The space required by the compressed index achieves the high-order empirical entropy of the string. In addition, we present approximate pattern matching on compressed suffix array for short read alignment. CS2A supports both mismatch and gapped alignments for single-end and paired-end reads mapping. The experiments show that CS2A can compete with the popular aligners in memory usage and mapping accuracy. The source code is available online [8].

1068-0314/16 $31.00 © 2016 IEEE
DOI 10.1109/DCC.2016.58

II. METHODS

A. Index structure

Let T be a string of n characters over an alphabet Σ of size σ and P a query pattern of length m. The suffix array $SA[0..n-1]$ of T is an array of n integers that gives the sorted order of the suffixes of T. $SA[i] = j$ means that suffix $T[j..n]$, starting at position j in T ranks the ith smallest among all the n suffixes. All the suffixes prefixed by P form a lexicographically contiguous range in the sorted array SA. Thus, we can use binary search to search for the range of SA containing the suffixes prefixed by P. The neighbor function $\Phi(i) = SA^{-1}[SA[i]+1]$ of the CSA [4] maps a position i in SA, such that $SA[i] = m$ for some m, into the position j, such that $SA[j] = m+1$. The values of Φ forms a piece-wise increasing sequence [4]. A practical index structure for the CSA of T is shown in Figure 1 [7], where $n = 36$, superblock size $a = 9$, and block size $b = 3$. "#" is a special end-of-text symbol smaller than any other character in T.

i	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19	20	21	22	23	24	25	26	27	28	29	30	31	32	33	34	35
T	a	t	a	g	t	t	a	t	g	t	a	c	c	t	g	a	c	g	a	c	g	g	c	t	g	a	g	t	a	c	a	t	t	g	a	#
SA	35	34	28	10	15	18	25	2	0	6	30	29	11	16	19	12	22	33	14	17	24	21	20	26	8	3	27	9	1	5	32	13	23	7	4	31
\Phi	8	0	11	12	13	14	23	25	28	33	35	10	15	19	22	31	32	1	4	5	6	16	21	26	27	34	2	3	7	9	17	18	20	24	29	30
gap	0	28	11	0	1	1	0	2	3	0	2	11	0	4	3	0	1	5	0	1	1	0	5	5	0	7	4	0	4	2	0	1	2	0	5	1
S	000011100000101 1			1 1			010 011			0100001011			00100 011			1 00101			1 1			0010100101			0011100100			00100 010			1 010			00101 1		
SB	0									24									48									70								
B	0			16			18			0			10			18			0			2			12			0			8			12		
SAM	8			12			23			33			15			31			4			16			27			3			17			24		

Figure 1. Index structure of T = atagttatgtacctgacgacggctgagtacattga#

We use formula (1) to access the values of Φ. $SAM[\lfloor i/b \rfloor]$ is the sampling array, which samples Φ at regular step b. *decompress* conducts a decoding on the encoded sequence S, which is obtained by applying Elias Gamma coding [3] to the gap sequence. Given position i, $SB[\lfloor i/a \rfloor] + B[\lfloor i/b \rfloor]$ is the decoding starting position. $i \mod b$ is the number of times to be decoded. We refer the reader to references [7] for an extensive description of the index structure.

$$\Phi[i] = (SAM[\lfloor i/b \rfloor] + decompress(S, SB[\lfloor i/a \rfloor] + B[\lfloor i/b \rfloor], i \mod b)) \mod n \quad (1)$$

Lemma 1 ([7]): Our index structure requires at most $2nH_k + n + o(n)$ bits of space in the worst case for any $k \leq c \log_\sigma n - 1$ and any constant $c < 1$, where H_k is the kth-order empirical entropy of T and σ is the alphabet size.

B. Pattern matching

Let $C[c]$ be the total number of occurrences in T of the characters that are alphabetically smaller than $c \in \Sigma$. That is, $C[c]$ denotes the number of suffixes in T whose leading character is less than c. Let P be a pattern of length m. We now show how to search for P in S, which is the compressed representation of T. Since the suffixes prefixed by P are lexicographically contiguous in the sorted array SA, thus, we can search for the range of SA containing the suffixes prefixed by P using binary search. The search determines the starting position l for the suffixes lexicographically larger than or equal to P and the ending position r for suffixes that start with P. We use $[l, r]$ to denote the suffix range to which P belongs. That is, $SA[l]$, $SA[l+1]$, $\cdots$, $SA[r]$ contains positions of these suffixes that are prefixed by P.

Algorithm 1 describes how to perform a match using the neighbor function Φ. We use pattern $P = \mathrm{gt}$ as an example to show how to use Algorithm 1 to perform pattern matching. Start from $c = p_1 = \mathrm{t}$ with $l_0 = 0$ and $r_0 = 35$, we have $l_c = C[\mathrm{t}] = 26$ and $r_c = C[\mathrm{t}+1] - 1 = 35$, thus $l = \min\{j : j \in [26, 35] \text{ and } \Phi(j) \in [0, 35]\} = 26$, $r = \max\{j : j \in [26, 35] \text{ and } \Phi(j) \in [0, 35]\}$ = 35. This shows that character t occurs $(35 - 26) + 1 = 10$ times in T. If we continue to call Algorithm 1 for next character $c = p_0 = \mathrm{g}$, we have $l_0 = 26$ and $r_0 = 35$, we have $l_c = C[\mathrm{g}] = 17$ and $r_c = C[\mathrm{g}+1]-1 = 25$, thus $l = \min\{j : j \in [17, 25] \text{ and } \Phi(j) \in [26, 35]\} = 23$, $r = \max\{j : j \in [17, 25] \text{ and } \Phi(j) \in [26, 35]\} = 25$. This shows that pattern $P = \mathrm{gt}$ occurs $(25 - 23) + 1 = 3$ times in T. c is a parameter of Algorithm 1 and can be instantiated a character when Algorithm 1 is called in Algorithms 2 and 3. If we want the compressed suffix array to work for the short read alignment, we have to modify the pattern matching algorithm to support approximate matching because reads may have errors, and the reference genome is usually not exact for the reads.

Algorithm 1 One match

Input: Φ, C, l_0, r_0, c
Output: l, r

OneMatch(Φ, C, l_0, r_0, c)
1: $l_c \leftarrow C[c]$; $r_c \leftarrow C[c+1] - 1$
2: $l \leftarrow \min\{j : j \in [l_c, r_c] \text{ and } \Phi(j) \in [l_0, r_0]\}$
3: $r \leftarrow \max\{j : j \in [l_c, r_c] \text{ and } \Phi(j) \in [l_0, r_0]\}$
4: **return** l and r

C. Approximate pattern matching

Let $P = p_0 p_1 \dots p_{m-1}$ be a short read of length m and $P_i = p_i \dots p_{m-1}$ is a suffix of P, where $0 \leq i \leq m-1$. Starting from p_{m-1}, assumed that we have found p_i and obtained a suffix range $[l_i, r_i]$ where the suffixes start with $p_i \dots p_{m-1}$. According to the backward search, the next step should search for p_{i-1} on the basis of $[l_i, r_i]$, and we would obtain a new range $[l_{i-1}, r_{i-1}]$ in T of suffixes starting with $p_{i-1} p_i \dots p_{m-1}$. However, if we do not find p_{i-1} but find a character $s \neq p_{i-1}$, we would obtain another suffix range $[l'_{i-1}, r'_{i-1}]$ instead. Then we may continue to search for $p_0 \dots p_{i-3} p_{i-2}$ backwards based upon $[l'_{i-1}, r'_{i-1}]$, until a range $[l_0, r_0]$ is obtained. $[l_0, r_0]$ is the suffix range of the approximate string of P, where p_i is substituted with another character s. Similarly, we can obtain an approximate string of P by deleting or inserting a character at position i.

Using substitute, delete or insert operation, we can align a short read onto a reference genome inexactly. The alignment procedure, however, could be time consuming. Since we do not know the exact positions where a substitute, insert or delete may occur; we need to try these operations on every possible position of a read. Each of the operations on a position would introduce a new approximate read. Before aligning all the approximate reads to the reference genome, we do not know which approximate read will result in the best alignment. Therefore, every approximate read could be a potential candidate and should be stored during the approximate matching.

Each operation in the approximate matching can be seen as a traversal in a search tree. For a short read P of length m, we can do four kinds of insert operations, three kinds of substitute operations, one kind of delete operation, and normal alignment at a position i. Therefore, each internal node in the search tree would have 9 children nodes; and the total number of branches in the search tree is 9^m in the worst case, which is computationally prohibitive.

D. Branch and Bound

In order to accelerate the search tree traversal for the approximate matching of a short read, we use two branch and bound strategies to prune branches of the search tree. First, we also use a difference array d, inspired by [11] to restrict the number of branches. $d[i]$ represents the maximum number of operations (replace, insert, or delete) allowed in the searching for $P_i = p_i \dots p_{m-1}$. It can be computed by Algorithm 2.

Algorithm 2 Difference array computation

Input: Φ, C, P
Output: d

Diff(Φ, C, P)
1: $z \leftarrow 0$; $l \leftarrow 0$; $r \leftarrow m-1$
2: **for** $i \leftarrow m-1$ down to 0 **do**
3: $(l, r) \leftarrow \text{OneMatch}(\Phi, C, l, r, p_i)$
4: **if** $l > r$ **then**
5: $l \leftarrow 0$; $r \leftarrow m-1$; $z \leftarrow z+1$ ▷ mismatch, increment z
6: **end if**
7: $d[i] \leftarrow z$
8: **end for**
9: **return** d

Second, we use the penalty strategy to limit the number of branches. During the search tree traversal for the alignment of the short read P of length m, performing any operation (substitute, insert or delete) will decrease the similarity between the approximate string corresponding to the search direction and P. We assign a different penalty score to insert, delete, and

substitute, respectively. We use the affine gap penalty for successive indels. When the penalty score of a search direction is greater than a given maximal penalty value $maxp$, we discard the direction, since it cannot drive a good solution.

E. Alignment algorithm

We create a priority queue data structure and change the depth-first search to breadth-first search in the search tree. We rank all search directions in increasing penalty score in each level and stored in the priority queue. With the priority queue structure, we can restrict the search to the optimal direction. In addition, we can discard the worst search direction by specifying the size of the priority queue though it satisfies bounded conditions.

Algorithm 3 Extension

Input: $P, \Phi, C, x, heap$
Output: $heap$

Extend($P, \Phi, C, x, heap$)

1: $y.l \leftarrow x.l;\ y.r \leftarrow x.r;\ y.i \leftarrow x.i - 1;\ y.z \leftarrow x.z + 1$ ▷ delete p_i
2: $y.pen \leftarrow x.pen + delPen;\ \text{insertHeap}(heap, y)$
3: **for all** $s \in \{a, c, g, t\}$ **do**
4: $(l, r) \leftarrow \text{OneMatch}(\Phi, C, X.l, X.r, s)$
5: **if** $l \leq r$ **then**
6: $y.l \leftarrow l; y.r \leftarrow r; y.i \leftarrow x.i;\ y.z \leftarrow x.z + 1$ ▷ insert s after p_i
7: $y.pen \leftarrow x.pen + insPen;\ \text{insertHeap}(heap, y)$
8: **if** $p_i = s$ **then**
9: $y.l \leftarrow l; y.r \leftarrow r; y.i \leftarrow x.i - 1;\ y.z \leftarrow x.z$ ▷ match
10: $y.pen \leftarrow x.pen;\ \text{insertHeap}(heap, y)$
11: **else**
12: $y.l \leftarrow l; y.r \leftarrow r; y.i \leftarrow x.i - 1\ y.z \leftarrow x.z + 1$ ▷ substitute
13: $y.pen \leftarrow x.pen + subPen; \text{insertHeap}(heap, y)$
14: **end if**
15: **end if**
16: **end for**
17: **return** $heap$

We use min binary heap to implement the priority queue. Each element keeps some information on the approximate matching. We take the corresponding penalty score and z value as key. Let x be a partial alignment of P associated with the suffix range $[l, r]$, penalty and the number of operations (substitute, insert or delete) performed so far. The extension procedure is described in Algorithm 3.

CS2A is described in Algorithm 4. CS2A first calculates difference array d of P, and initialize a partial alignment entity x with $l = 0, r = n - 1, i = m - 1, pen = 0$ and $z = 0$, and two data structures $heap$ and $result$. The $heap$ is used to store intermediate alignment results. The $result$ stores the final alignment that satisfies alignment constraints. During the alignment, we first extract a partial alignment x with the smallest penalty pen from the priority queue $heap$. If the current alignment position $i < 0$, which means that x has been aligned to p_0, we identify x as a reasonable alignment and insert x into the priority queue $result$, then we carry out next iteration; otherwise, we handle x according to branch and bound strategy.

III. Implementations

Based on the alignment algorithm proposed above, we implement a short read aligner: CS2A. CS2A aligner consists of three procedures: $Buildindex$, $Alignment$, and $Output$. $Buildindex$ creates the index structure for the reference genome T. $Alignment$ is the core of CS2A, which aligns short reads onto T. The alignment result is stored in an intermediate file format: SAI. $Output$ transforms SAI to standard SAM (Sequence Alignment/Map) format [13]. In order to improve alignment efficiency and accuracy, we also make further optimizations on the CS2A: seed policy and paired-end mapping.

A. Creating the index for the reference genome

Because we do not know from which DNA strand a short read is sequenced, we need to align the short read against both forward strand and reverse strand of a reference genome. In order to reduce alignment time, we construct a new reference

Algorithm 4 CS2A alignment

Input: $heap, \Phi, C, P, maxP, maxHeapSize$
Output: $result$

CS2A($heap, \Phi, C, P, maxP, maxHeapSize$)
1: $d \leftarrow \text{Diff}(\Phi, C, P)$
2: $heap \leftarrow \varnothing; result \leftarrow \varnothing$
3: $x.l \leftarrow 0; x.r \leftarrow n-1; x.i \leftarrow m-1; x.pen \leftarrow 0; x.z \leftarrow 0$
4: $heap \leftarrow$ Extend($P, \Phi, C, x, heap$) ▷ handle p_{m-1}
5: **while** $heap \neq \varnothing$ **do**
6: $x \leftarrow$ extractMin($heap$)
7: **if** $x.i < 0$ **then**
8: Insert($result, x$); continue ▷ obtain a good alignment
9: **end if**
10: **if** $x.z \leq d[x.i]$ and $x.pen \leq maxP$ **then**
11: $heap \leftarrow$ Extend($P, \Phi, C, x, heap$) ▷ handle $p_0 \ \ldots \ p_{m-2}$
12: **end if**
13: **while** heapSize($heap$) $> maxHeapSize$ **do**
14: heapDropMax($heap$)
15: **end while**
16: **end while**
17: **return** $result$

genome by connecting the reference genome and its complementary strand together. So, we need to align a short read to reference genome only once.

Given a string T, to construct its compressed suffix array we need to construct the suffix array of T firstly. However, the suffix array construction usually occupies about 9 times memory footprint of the original text size. Considering the scale of the human genome (2.8 GB), this approach is difficult to be used for constructing the suffix array of genomic sequences with a normal PC. Therefore, we use a space-efficient construction method proposed in [6] to construct the compressed suffix array of the reference genome. As the CSA of the reference genome is created only once, it can be re-used.

B. Using the SAI intermediate file

Using CS2A, we can obtain a suffix array interval $[l, r]$ of a pattern P quickly. However, it is time-consuming for the CS2A to map the interval $[l, r]$ to chromosome coordinates. We define the Sequence Alignment Intermediate file format (SAI) to save the alignment time, which is used to store intermediate alignment results. The SAI file only stores short read alignments occurring in the suffix range $[l, r]$, mapping accuracy [11] and performed operations information. Combining with the index structure of the reference genome, the Output procedure transforms the intermediate alignment results from the SAI format to the SAMformat [13].

C. Using seed to improve accuracy

In DNA sequencing, the closer a base is to the $5'$ end of a DNA fragment, the higher its call quality is, and vice versa. In the seed-and-extend alignment approach, the high-quality end of a short read is selected as seed, which only allows one or two mismatches in alignment. Using the seed policy, we can further improve search accuracy and efficiency. In detail, we select the first 32 bases from $5'$ end of a short read as seed by default. The number of mismatches, gap openings, and gap extensions has more limits on the seeded than non-seeded subsequences. Because of the high limit in the seed region, we can align seed firstly and discard search directions which do not satisfy the constraint of seed policy.

D. Paired-end mapping

CS2A supports paired-end alignment. Paired-end sequencing technique sequences both ends of a DNA fragment and outputs a pair of reads. Given a pair of short reads, we align the two reads using CS2A respectively, and then we check the two sets of alignment results to find consistent pairs.

Restricted by sequencing accuracy, paired-end sequencing can only sequence two ends of a DNA fragment. Usually, there is an area between a pair of reads where it cannot be sequenced. Furthermore, the exact length of a DNA fragment is unknown. The default fragment length of Illumina sequencer is between 200 and 300 bps. In paired-end mapping, when

a pair of reads are aligned to different strands of the reference genome, and their genomic coordinates are within a given maximal distance, it is considered as a consistent pair.

IV. Experimental results

A. Data Sets and Settings

We made a comparison of CS2A with three popular aligners: Bowtie [10], BWA [11], and SOAP2 [17]. All the aligners are carried out on a workstation that has 24 Intel(R) Xeon(R) CPU E5-2692 v2 @2.20GHz cores and 64GB of memory. The operating system is Red Hat Enterprise Linux Server release 6.4 (Santiago).

Efficiency and accuracy are two important measurements of short read alignment. Besides alignment time and memory footprint, we use Confidence and Mapping accuracy [11] to evaluate the performance of different aligners. Confidence is defined as the fraction of confidently mapped reads to all the reads (Conf), calculated by S/N and mapping accuracy is defined as the fraction of confidently and correctly mapped reads among all the confidently mapped reads (Acc), calculated by S_1/S, where N is the number of total reads, S is the number of confidently mapped reads, and S_1 is the number of confidently and correctly mapped reads.

In the experiments, we use the human genome 'hg19.fa' released by NCBI as the reference genome. Using the 'wgsim' simulator in SAMtools [13], we generate three paired-end short read samples from the reference genome, including ds35, ds70 and ds125. Each sample is composed of one million pairs of reads. The read length of ds35, ds70 and ds125 is 35, 70 and 125, respectively. The SNP mutation rate of 'wgsim' is set to 0.09%, indel mutation rate is set to 0.01%, and fragment distance of a pair of reads meets the normal distribution N(500, 50).

B. Influence of the seed length on performance

Firstly, we analyze the influence of seed length on alignment accuracy and efficiency. Suppose that this is an unspaced seed. With different seed lengths, we map ds70 to the reference genome using CS2A. The experimental results are shown in Figure 2. It shows that the alignment time decreases with the increasing of seed length. The mapping accuracy increases with the increasing of seed length, but the Confidence decreases. This is because with longer seed CS2A can filter out more candidate alignments at the beginning. When setting seed length to 32bp, we can align one million pairs of 70bp reads in 1.272 hours, and get nearly 85% Conf and more than 99.9% Acc. To strike a balance among alignment time, Confidence and Accuracy, we set the seed length to 32bp without specific statement in the following experiments.

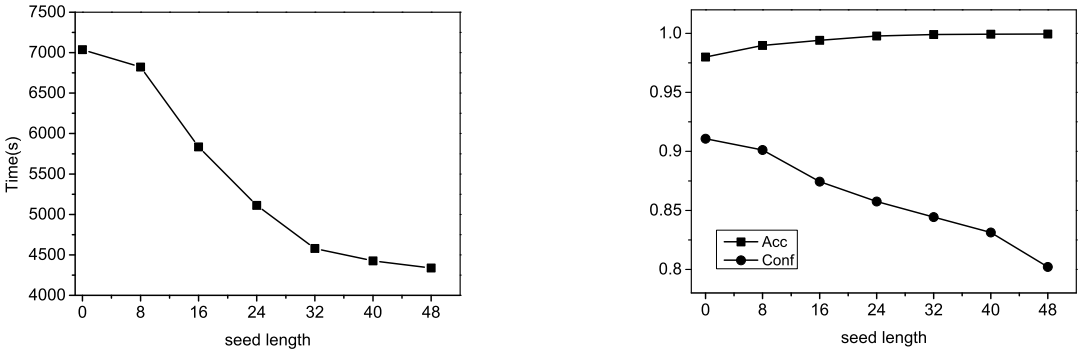

Figure 2. The alignment effect of CS2A under different seed lengths

C. Evaluation on simulated data sets

To compare the performance of these aligners, we align ds35, ds70 and ds125 short read samples to the human genome 'hg19.fa' in single-end and paired- end modes. All the aligners are deployed using their default configuration. Their mapping accuracies are evaluated using wgsim_eval.pl included in the SAMtools [13] package. The experimental results are given in Table I, where T, M, C, and A represents time (in hour), memory (in GB), confidence, and mapping accuracy, respectively. In column 'Program' of Table I, the number after aligner name gives the read length. For example, Bowtie-35 means that using Bowtie to map ds35 to the human genome hg.19.

Table I shows that in both single-end and paired-end alignment modes, the time required by each aligner increases with the increase of read length. For all data sets, SOAP2 is among the fastest alignment but has the worst memory usage. SOAP2

can compete in Conf and Acc with the other three methods. Bowtie is among the smallest space usage. Both CS2A and BWA can also map the simulated samples in acceptable amounts of time. The Confidence of CS2A is better than Bowtie in both single-end and paired-end mapping, which means that CS2A can correctly align more reads than Bowtie, but slightly less than BWA; at the same time, CS2A and BWA have similar levels of accuracy, both are more accurate than Bowtie. The memory usage of CS2A is between those of Bowtie and BWA.

Table I
SINGLE/PAIRED-END MAPPING ON SIMULATED DATA SETS

Program	Single-end mapping				Paired-end mapping			
	T	M	C(%)	A(%)	T	M	C(%)	A(%)
Bowtie-35	0.292	2.85	77.55	97.06	0.298	2.85	81.33	98.54
BWA-35	0.256	3.41	80.32	99.76	0.420	4.76	90.48	99.94
SOAP2-35	0.04	5.56	82.10	99.13	0.20	5.57	91.82	99.73
CS2A-35	0.514	3.25	79.88	99.74	0.539	3.79	83.35	99.93
Bowtie-70	0.411	2.85	80.21	98.64	0.441	2.86	83.47	98.67
BWA-70	0.666	3.42	89.33	99.91	0.724	4.84	94.71	99.97
SOAP2-70	0.09	5.59	89.32	99.64	0.18	5.60	92.25	99.78
CS2A-70	1.272	3.30	84.44	99.91	1.299	3.93	88.57	99.97
Bowtie-125	0.497	2.86	83.36	99.27	0.534	2.88	84.12	99.02
BWA-125	1.689	3.44	91.57	99.96	1.729	4.91	96.11	99.99
SOAP2-125	0.25	5.61	89.73	99.87	0.55	5.61	83.76	99.85
CS2A-125	3.438	3.37	86.83	99.97	3.473	4.06	90.96	99.96

D. Evaluation on real-world data sets

To evaluate the performance of CS2A on real-world short reads data sets, we download 12.2 million pairs of 51bps reads from European Read Archive (AC: ERR000589) and align these short reads to the hg19.fa human reference genome. The experimental results are given in Table II, where Time is in hour and Memory is in GB. Similar to the results obtained on the simulated samples, all four aligners can align these reads to the reference genome in acceptable amounts of time. SOAP2 is among the fastest alignment but has the worst memory usage. BWA and SOAP2 align more reads than Bowtie and CS2A, in the meanwhile almost all reads mapped by BWA and CS2A can be paired. The memory usage of CS2A is smaller than that of BWA, but larger than that of Bowtie.

Table II
EVALUATION ON REAL-WORLD DATA SETS

Program	Time	Memory	Conf(%)	Paired(%)
Bowtie	4.08	2.94	84.71	95.43
BWA	3.08	4.73	87.67	99.83
SOAP2	2.20	5.56	88.81	98.04
CS2A	6.83	3.91	83.18	99.75

V. CONCLUSION

Next generation sequencing technologies greatly enable the studying of gene expression and genome variation on a genome scale. Although there has been several alignment tools available, short reads alignment still is one of the major bottlenecks in the analysis of NGS Data. In this paper, we implemented a novel short reads aligner: CS2A, which is based upon CSA and backward search. It supports gapped alignment for single-end reads and paired-end mapping. The experiments on both simulated and real-world data show that CS2A can map millions of short reads in acceptable amounts of time; the memory footprint of CS2A is independent of the scale of short reads, and is less than 4GB for the human genome; the mapping accuracy of CS2A is comparable to the state-of-the-art aligners.

Although this work shows that CS2A can align NGS short reads onto a large reference genome such as the human genome, there is still room for future development. Firstly, CS2A only maps nucleotide-space short reads generated by Illumina or Roche 454 systems. It should be extended to map color-space short reads obtained from the ABI SOLiD system. Secondly, CS2A does not completely use base call quality scores reported in the FASTQ files. Higher sensitivity is expected to be achieved if we set different penalty values for different quality bases in the branch and bound strategy. Finally, for simplicity CS2A indiscriminately converts any nucleotide that is not 'A', 'C', 'G' or 'T' to 'N'. Since this may decrease alignment accuracy, future work will also include considering the specific meaning of such ambiguous but nevertheless informative nucleotides.

ACKNOWLEDGMENT

The authors would like to thank Heng Li for clarifying some data structures related to BWA [11], and also wish to thank the anonymous reviewers for their careful reading and their constructive comments, which have considerably improved the quality of the paper. This work is supported in part by China NSF grants 61173025 and 61373044 (H. Huo), and China NSF grant 61502366 (Q. Yu), by US NSF grant CCF-1017623 (J.S. Vitter), and US NIH grants P20 GM103638, P30 AG035982, and P30 HD002528 (X. Wang).

REFERENCES

[1] C. Alkan and J. M. Kidd. Personalized copy number and segmental duplication maps using next-generation sequencing. *Nature Genetics*, 41(10):1061–1067, 2009.

[2] M. Burrows and D. J. Wheeler. A block-sorting lossless data compression algorithm. Technical Report 124, Digital Equipment Corporation, Palo Alto, CA, USA, 1994.

[3] P. Elias. Universal codeword sets and representations of the integers. *IEEE Transactions on Information Theory*, 21(2):194–203, 1975.

[4] R. Grossi and J. S. Vitter. Compressed suffix arrays and suffix trees with applications to text indexing and string matching. *SIAM Journal on Computing*, 35(2):378–407, 2005.

[5] R. Grossi, A. Gupta, and J. S. Vitter. High-order entropy-compressed text indexes. In *Proceedings of Symposium on Discrete Algorithms*, pages 841–850, 2003.

[6] W. K.Hon, T. W.Lam, K. Sadakane, et al.. A space and time efficient algorithm for constructing compressed suffix arrays. *Algorithmica*, 48(1): 23C-36, 2007.

[7] H. Huo, L. Chen, J. S. Vitter, and Y. Nekrich. A Practical Implementation of Compressed Suffix Arrays with Applications to Self-Indexing. In *Proceedings of Data Compression Conference*, pages 292–301, 2014.

[8] H. Huo, Z. Sun, S. Li, J. S. Vitter, X. Wang, Q. Yu, and J. Huan. Source code for compressed suffix array-based short read alignment. https://github.com/bluecliff/csa-alignment, 2014.

[9] O. M. Külekci, W. K. Hon, R. Shah, J. S. Vitter, and B. Xu. Ψ-RA: a parallel sparse index for genomic read alignment. *BMC Genomics*, 12(Suppl 2):S7, Jul. 2011.

[10] B. Langmead, C. Trapnell, M. Pop, and S. Salzberg. Ultrafast and memory-efficient alignment of short DNA sequences to the human genome. *Genome Biology*, 10(3):R25, 2009.

[11] H. Li and R. Durbin. Fast and accurate short read alignment with Burrows-Wheeler transform. *Bioinformatics*, 25(14):1754–1760, 2009.

[12] H. Li, J. Ruan, and R. Durbin. Mapping short DNA sequencing reads and calling variants using mapping quality scores. *Genome Research*, 18(11):1851–1858, 2008.

[13] H. Li, B. Handsaker, A. Wysoker, T. Fennell, J. Ruan, N. Homer, et al.. The sequence alignment/map format and SAMtools. *Bioinformatics*, 25(16):2078–2079, 2009.

[14] H. Li and H. Nils. A survey of sequence alignment algorithms for next-generation sequencing. *Briefings in Bioinformatics*, 11(5):473–483, 2010.

[15] Y. Li, J. M. Patel, and A. Terrell. Wham: a high-throughput sequence alignment method. *ACM Transactions on Database Systems*, 37(4):28, 2012.

[16] R. Li, Y. Li, K. Kristiansen, and J. Wang. SOAP: short oligonucleotide alignment program. *Bioinformatics*, 24(5):713–714, 2008.

[17] R. Li, C. Yu, Y. Li, T. W. Lam, S. M. Yiu, K. Kristiansen, and J. Wang. SOAP2: an improved ultrafast tool for short read alignment. Bioinformatics, 25(15):1966–1967, 2009.

[18] Y. Liao, G. Y. Smyth, and W. Shi. The Subread aligner: fast, accurate and scalable read mapping by seed-and-vote. *Nucleic Acids Research*, gkt214, 2013.

[19] M. L. Metzker. Sequencing technologies - the next generation. *Nature Reviews Genetics*, 11(1):31–46, 2010.

[20] M. Ruffalo, T. LaFramboise, and M. Koyutürk. Comparative analysis of algorithms for next-generation sequencing read alignment. *Bioinformatics*, 27(20):2790–2796, 2011.

[21] G. Zhang, T. Fedyunin, S. Kirchner, C. Xiao, A. Valleriani, and Z. Ignatova. FANSe: an accurate algorithm for quantitative mapping of large scale sequencing reads. *Nucleic Acids Research*, gks196, 2012.