Memory-efficient Short Read Sequence Alignment Algorithm for Human Genome Sequence

Shovan Roy, Sunirmal Khatua

Abstract— Any genetic change in a population that is inherited over several generations is called **Biological evolution**. Biological evolution is a scientific theory that was proposed by **Charles Darwin** in "**The Origin of Species by Means of Natural Selection**". Today we need to build a relation between the development of methods for the management and analysis of biological information arising from genomics and high-throughput experiments. The development of computational methods for studying the structure, function, and evolution of genes, proteins, and whole genomes form the **bioinformatics**. High-throughput experiments produce large amounts of quantitative data. This poses challenges for bio-informaticians. How do we store information is such a way that it can be compared with results from others? How do we best extract meaningful information from the vast amount of data? New methods are needed to spot significant trends & patterns in the data. This is a new area of biological sciences where computational methods are essential for the progress of the experimental science where algorithms & experimental techniques are being developed side by side.

Index Terms— Read, Alignment, BWT, L-F Mapping, Bowtie, Exact Matching, Inexact Matching, MAQ, Full Bowtie Algorithm.

1 Problem Description

In March 1953, Watson and Crick deduced the double helix structure of DNA, thus proving that it carried the genetic information. The challenge of reading the DNA sequence became central dogma to biological research era. The earliest chemical methods for DNA sequencing were extremely not fully capable, laborious and costly. Over the next few decades, sequencing became more efficient by orders of magnitude. In the 1970s, two classical methods for sequencing DNA fragments were developed by Sanger and Gilbert. In the 1980s beside these methods, cloning method allowed fast and exponential replication of a DNA fragment. The human genome project was started in the 1990s, sequencing efficiency had already reached 200,000 bp/person/year, and when it concluded in 2002 this figure had gone up to 50,000,000 bp/person/year. Then new sequencing technologies have emerged, which now allow the reliable sequencing of $100x10^9$ bp/person/year. At the same time, the cost of sequencing has also sharply declined.

Sketch the sequencing technology [16, 17, and 18]: There are two types of nucleic acid in cells- DNA (Deoxyribonucleic Acid) and RNA (Ribonucleic Acid). DNA is a polymer containing chains of nucleotide monomers. Each nucleotide contains a sugar, a base, and a phosphate group. The sugar is 2′-deoxyribose which has five carbons named 1′ (prime) 2′ etc. There are four types of base: Adenine (A) and Guanine (G) have two carbon-nitrogen rings and are Purines; Thymine (T) and Cytosine (C) have a single ring and are pyrimidines. A Sugar + A Base = Nucleoside. A nucleotide has one, two or three phosphate groups attached to the 5′ carbon of the sugar. A Sugar + A Base + A Phosphate Group = Nucleotide. In RNA thymine (T) is replaced by uracil (U) & 2-deoxyribose by ribose. It is a single polynucleotide strand. A, G and T, C form the sequence termed as read and they form the input for the computational problems.

We first focus on the problem of aligning the reads to the genome.

Problem: Short read mapping problem.
Input: m l-long reads $S_1, \ldots, S_m$ and an approximate reference genome R.
Question: What are the positions $x_1, \ldots, x_m$ along R where each read matches?

An example of this problem is when we sequence the genome [15] of a person and wish to map it to an existing sequence of the human genome. The new sample will not be 100% identical to the reference genome due to the natural variation in the population, and so some reads may align to their position in the reference with mismatches or gaps. In diploid organisms, such as human beings, different alleles on the maternal and paternal chromosomes can lead to two slightly different reads mapping to the same location (some perhaps with mismatches). Additional complications may arise due to sequencing errors or repetitive regions in the genome which make it difficult to decide where to map the read.

2 Possible Solutions of the Above Mapping Problem

2.1 Local Alignment [4, 8, 12]

The problem of aligning a short read to a long genome sequence is exactly the problem of local alignment. However, the large parameters involved make such an approach impractical. In the human genome example, the number of reads m is usually 10^7-10^8, the length of a read l is 50-200 bp and the length of the genome $|R|$ is 3.10^9 bp (or twice, for the diploid genome).

- *Shovan Roy, M. Tech in Computer Science and Engineering in University of Calcutta, India, Mob. No.: 9830656545. E-mail: sho.cmsa.08@gmail.com*
- *Sunirmal Khatua, Assistant Professor, Department of Computer Science and Engineering, University College of Science and Technology, University of Calcutta, Kolkata, 92, A. P. C. Road, Kolkata-700009, India, Ph. No.: 033-25594177, Mob. No.: 9874872149. E-mail: skhatuacomp@caluniv.ac.in*

Let us consider some other possible solutions:

I. The most naive algorithm would scan R for each S_i, matching the read at each position p and picking the best match. Time complexity: O (ml|R|) for exact or inexact matching. Considering the parameters mentioned above, this is clearly impractical.

II. A less naive solution uses the Knuth-Morris-Pratt algorithm (KMP algorithm) to match each S_i to R.

Time complexity: O (m (l+|R|)) = O (ml + m|R|) for exact matching. This is a substantial improvement but still not enough.

III. Building a suffix tree [5] for R provides another solution. Then, for each S_i we can find matches by traversing the tree from the root. Time complexity: O(ml+|R|), assuming the tree is built using Ukkonen's linear-time algorithm. This time complexity is certainly practical, and it has the additional advantage that we only need to build the tree for the reference genome once. It can then be saved and used repeatedly for mapping new sequences, at least until an updated version of the genome is issued.

However, space complexity now becomes the obstacle the leaves of the suffix tree also hold the indices where the suffixes begin, saving the tree requires O (|R|log |R|) bits just for the binary coding of the indices, compared with |R|log |R|) bits for the original text. The constants are also large due to the additional layers of information required for the tree (such as suffix links, etc.). Thus, we can store the text of the human genome using ~750MB, but we'd need ~64GB for the tree! The resultant size is much greater than the cache memory of most of today's desktop computers. Another problem is that suffix trees allow only for exact matching.

IV. A fourth solution is to preprocess the reference genome into a hash table H. The keys of the hash are all the substrings of length l in R, and the value of each key is the position p in R where the substring ends. Then, given S_i the algorithm returns H (S_i).

Time complexity: O (ml + l|R|), which is pretty good. The space complexity, however, remains too high at O (l |R| + |R| log |R|) since we must also hold the binary representation of each substring's position. A practical improvement which can be applied is packing the substrings into bit-vectors, that is representing each nucleotide as a 2-bit code. This reduces the space complexity by a factor of four. Further improvement can be achieved by partitioning the genome into several chunks, each of the size of the cache memory, and running the algorithm on each chunk in turn. Again, this approach only allows exact matching.

2.2 What is my Target?

We have seen that one way to map reads to a reference genome is to index into a hash table either all l-long windows of the genome or of the reads. Holding these indices in memory requires a great deal of space, as discussed in above section. I suggest the Bowtie algorithm [1] to solve this problem.

2.3 Definition and Problem Formulation

The Bowtie algorithm, presented in 2009 by Langmead [1], solves problem through a more space-efficient indexing scheme. This scheme is called the Burrows-Wheeler transform [2] and was originally developed for data compression purposes. In the following section, we will describe the transform and its uses by following a specific example.

2.4 Implementation

2.4.1 Burrows-Wheeler Transform (BWT) [2, 11]

BWT originally designed for data compression for large text. The Burrows-Wheeler transformation of a text T, BWT (T), is constructed as follows:

I. The character $ is appended to T, where $ is not in T and is lexicographically less than all characters in T.

II. The Burrows-Wheeler matrix of T is constructed as the matrix whose rows comprise all cyclic rotations of T$.

III. The rows are then sorted lexicographically.

IV. BWT (T) is the sequence of characters in the rightmost column of the Burrows- Wheeler matrix. BWT (T) has the same length as the original text T.

To demonstrate the process, we shall apply the transform BWT (T) to T="the_next_text_that_i_index.":

I. First, we generate all cyclic shifts of T.

II. Next, we sort these shifts lexicographically. /* In this example we define the character '.' as the minimum and we assume that it appears exactly once, as the last symbol in the text. It is followed lexicographically by '.', which is followed by the English letters according to their natural ordering. We call the resulting matrix M. */

We now define the transform BWT (T) as the sequence of the last characters in the rows of M. /* Figure1 shows an example for the first few shifts. Note that this last column is a permutation of all characters in the text since each character appears in the last position in exactly one cyclic shift. */

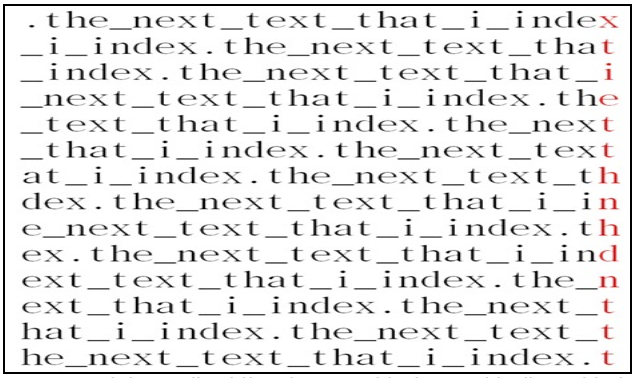

Fig. 1. Some of the cyclic shifts of T sorted lexicographically and indexed by the last character.

Saving BWT (T) requires the same space as the size of the text

T since it is simply a permutation of T. In the case of the human genome, we saw that each character can be represented by 2 bits, so we require ~$2.3.10^9$ bits for storing the permutation instead of ~$30.3.10^9$ for storing all indices of T.

Thus far, we have seen how to transform a text T into BWT (T). Let us now consider what information can be gleaned from BWT (T), assuming we do not see T or M:

I. The first question we can ask is: given BWT (T), how many occurrences in T of the character 'e'?
We can easily answer this by counting the number of occurrences of 'e' in BWT (T) since we have shown that this is simply a permutation of the text.

II. Can we also recover the first column of the matrix M?
Certainly! All we have to do is sort BWT (T) since the first column is also a permutation of all characters in the text, sorted lexicographically. Figure2 demonstrates this.

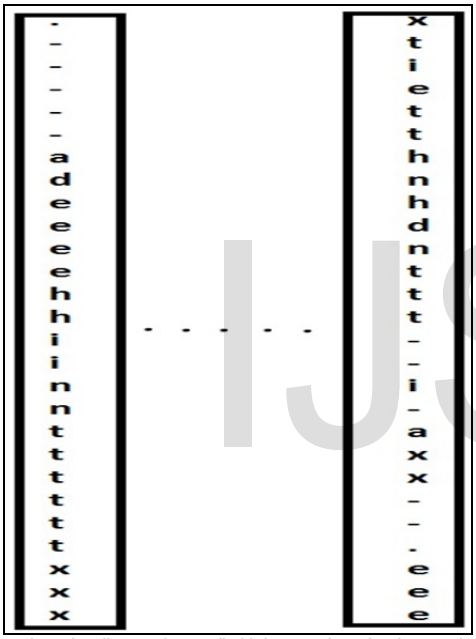

Fig. 2. Recovering the first column (left) by sorting the last column.

III. How many occurrences of the substring 'xt' do we have in T?
BWT (T) is the last column of the lexicographical sorting of the shifts. Hence, the character at the last position of a row appears in the text T immediately prior to the first character in the same row (each row is a cyclical shift). So, to answer this question, we consider the interval of 't' in the first column, and check whether any of these rows have an 'x' at the last position. In Figure 2, we can see that there are two such occurrences.

IV. Now we can recover the second column as well.
We know that 'xt' appears twice in the text, and we see that 3 rows start with an 'x'. Two of those must be followed by a 't', but which ones? The lexicographical sorting determines this as well. In the above example, another 'x' is followed by a '.' (see first row). Therefore, '.' must follow the first 'x' in the first column since '.' is smaller lexicographically than 't'. The second and third occurrences of 'x' in the first column are therefore followed by 't'. We can use the same process to recover the characters at the second column for each interval, and then the third, etc.

We have thus shown two central properties of the transform, which we now state formally following a formulation by **Ferragina** and **Manzini [3]**.

Lemma1 (Last-First Mapping):

Let M be the matrix whose rows are all cyclical shifts of T sorted lexicographically, and let L(i) be the character at the last column of row i and F(i) be the first character in that row.

Then:

I. The ith row of M, its last character L[i] precedes its first character F[i] in the original text T, namely T =…L(i) F(i)….

II. The j-th occurrence of character X in L corresponds to the same text character as the j-th occurrence of X in F. Let L[i] = c and let ri be the rank of the row M[i] among all the rows ending with the character c. Take the row M[j] as the ri-th row of M starting with c. Then the character corresponding to L[i] in the first column F is located at F[j] (we call this **LF-mapping**, where LF[i] = j).

Proof:

I. Follows directly from the fact that each row in M is a cyclical shift.

II. Let Xj denote the j-th occurrence of char X in L, and let α be the character following Xj in the text and β the character following Xj+1. Then, since Xj appears above Xj+1 in L, α must be equal or lexicographically smaller than β. This is true since the order is determined by lexicographical sorting of the full row and the character in F follows the one in L (property 1). Hence, when character Xj appears in F, it will again be above Xj+1, since α and β now appear in the second column and $X\alpha \leq X\beta$ (Figure 3 demonstrates this).

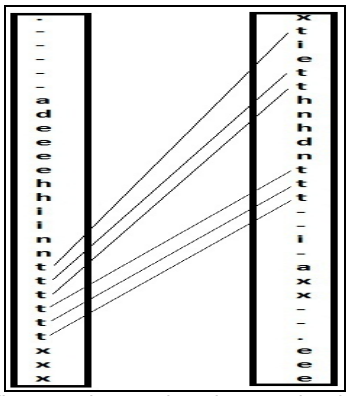

Fig. 3. Last-first mapping. Each 't' character in L is linked to its position in F and no crossed links are possible.

2.4.2 Reconstructing the Text

We now present an algorithm for reconstructing a text T from its Burrows-Wheeler transform BWT (T) utilizing Lemma1. In this formulation, we assume the actual text is of length u and we append a unique $ character at the end, which is the smallest, lexicographically (the '.' character played the role of $ in our example above).

We are therefore ready to describe the **backward BWT**:

UNPERMUTE [BWT(T)]:

I. Compute the array C[1,…, |∑|] : C(c) is the no. of characters {$,1,….,c-1} in T.

II. Construct the last-first mapping, tracing every character in L to its corresponding position in F:
LF[i] = C(L[i]) + r(L[i], i) + 1, where r(c, i) is the number of occurrences of character c in the prefix L[1, i].

III. Reconstruct T backwards as follows: s = 1, T(u) = L[1]; (because M[1] = $T); then, for each i = u-1, . . . ,1 do s = LF[s] and T[i] = L[s].

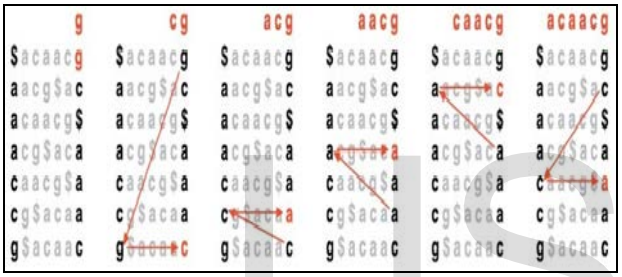

Fig. 4. Steps taken by EXACTMATCH to identify the range of rows, and thus the set of reference suffixes, prefixed by 'aac'. Source: [1].

In the above example of Fig. 4, T = acaacg$ (u = 6) was transformed to BWT (T) = gc$aaac, and we now wish to reconstruct T from BWT (T) using **UNPERMUTE**:

I. First, the array C is computed. /* For example, C(c) = 4 since there are 4 occurrences of characters smaller than 'c' in T (in this case, the '$' and 3 occurrences of 'a'). Notice, that C(c) + 1 = 5 is the position of the first occurrence of 'c' in F. */

II. Second, we perform the LF mapping. /* For example, LF[c2] = C(c) + r(c,7) + 1 = 6, and indeed the second occurrence of 'c' in F sits at F[6]. */

III. Now, we determine the last character in T: T(6) = L(1) = 'g'.

IV. We iterate backwards over all positions using the LF mapping. /* For example, to recover the character T(5), we use the LF mapping to trace L(1) to F(7), and then T(5) = L(7) = 'c'. */

Remark We do not actually need to hold F in memory, which would double the space we use. Instead, we only keep the array C defined above, of size |∑|, which we can easily obtain by looking at L alone.

2.4.3 Exact Matching

Next, we present an algorithm for exact matching of a query string P to T given BWT (T). The principle is very similar to UNPERMUTE, and we use the same definitions presented above for C and r(c, i). We denote by sp the position of the first row in the interval of rows in M we are currently considering, and by ep the position of the first row beyond this interval. So, the interval is defined by the rows sp ,…, ep - 1.

EXACTMATCH [P[1, . . . , p], BWT (T)]

1. c = P[p]; sp = C[c] + 1; ep = C[c+1] + 1; i = p - 1;
2. while sp < ep and i ≥ 1
   c = P[i];
   sp = C[c] + r(c, sp) + 1;
   ep = C[c] + r(c, ep) + 1;
   i = i - 1;
3. if(sp == ep) return "no match";
   Else
   return sp, ep;

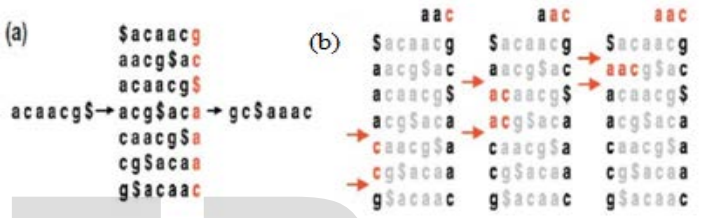

Fig. 5. **(a)** The Burrows-Wheeler matrix and transformation for 'acaacg'. **(b)** UNPERMUTE repeatedly applies the last first (LF) mapping to recover the original text (in red on the top line) from the Burrows-Wheeler transform (in black in the rightmost column). Source: [1].

In the above example of Fig. 5 we use the same text as in Figure 4, while searching for P = 'aac':

1. First, we initialize sp and ep to define the interval of rows beginning with the last character in P, which is 'c':
   sp = C(c) + 1 = 5.
   ep = C(g) + 1 = 7.
2. Next, we consider the preceding character in P, namely 'a'. We redefine the interval as the rows that begin with 'ac' utilizing the LF mapping. Specifically:
   sp = C(a) + r(a, 5) + 1 = 1 + 1 + 1 = 3.
   ep = C(a) + r(a, 7) + 1 = 1 + 3 + 1 = 5.
3. Now, we consider the preceding character, namely 'a'. We now redefine the interval as the rows that begin with 'aac'. Specifically:
   sp = C(a) + r(a, 3) + 1 = 1 + 0 + 1 = 2.
   ep = C(a) + r(a, 5) + 1 = 1 + 1 + 1 = 3.
4. Having thus covered all positions of P, we return the final interval calculated (whose size equals the number of occurrences of P in T).

Note that EXACTMATCH returns the indices of rows in M that begin with the query, but it does not provide the offset of each match in T. If we kept the position in T corresponding to the start of each row we would waste a lot of space. Instead, we can mark only some rows with pre-calculated offsets.

Then, if the row where EXACTMATCH found the query has this of set, we can return it immediately. Otherwise, we can successively use LF mapping to find a row that has a precalculated of set, and then we simply need to add the number of times we applied this procedure to obtain the position in which we are interested. There is a simple time-space tradeoff associated with this process.

Example In figure 5 we found the row beginning with 'aac'. Assuming that row has no offset, we can use LF mapping to reach row 5. If that row has the offset 2, then the desired offset of 'aac' is 2 + 1 = 3.

Note also that we did not describe how to efficiently compute r(c, i), which is an operation we repeat many times while running the algorithm. Again, it would be wasteful to save the value for each occurrence of every character in the text. Instead, we can use a similar solution to that used above for finding the exact index of a match. We store only a subset of the values and locally compute back from an unknown value to a stored one. Ferragina and Manzini provide a more efficient (and complicated) solution for this issue [3].

2.4.4 Inexact Matching

We have seen how to find exact matches of a query using BWT (T). However, to map reads to the genome we must allow for **mismatches**. Each character in a read has a numeric quality value, with lower values indicating a higher likelihood of a sequencing error. **Bowtie defines an alignment policy that allows a limited number of mismatches and prefers alignments where the sum of the quality values at all mismatched positions is low.** The search proceeds similarly to EXACTMATCH, calculating matrix intervals for successively longer query suffixes. If the range becomes empty (a suffix does not occur in the text), then the algorithm may select an already-matched query position and substitute a different base there, introducing a mismatch into the alignment. The EXACTMATCH search resumes from just after the substituted position. The algorithm selects only those substitutions that are consistent with the alignment policy and that yield a modified suffix that occurs at least once in the text. If there are multiple candidate substitution positions, then the algorithm greedily selects a position with a maximal quality value. Figure 6 demonstrates this. In the full Bowtie algorithm, backtracking can allow more than one mismatch, but the size of the backtracking stack is bounded by a parameter for efficiency.

3 Double Indexing Method to Solve Excessive Backtracking [1]

Excessive backtracking occurs in some cases for alignments a sequence. This occurs when the aligner spends most of its effort fruitlessly backtracking to positions close to the 3' end of the query. Bowtie mitigates excessive backtracking with the novel technique of '**double indexing**'. Two indices of the genome are created: one containing the **BWT of the genome**, called the '**forward**' **index** and a second containing the **BWT**

Fig. 6. Example of running the inexact match variant of the Bowtie algorithm. In this example, we try to map the string 'ggta' to the genome, but we only succeed at mapping 'ggtg'. The array at each level of the backtracking shows the row intervals corresponding to suffixes with the 4 nucleotides at that position (in the order a, c, g, t). Source: **[1].**

of the genome with its character sequence reversed (not reverse complemented) called the '**mirror**' **index**. To see how this helps, consider a matching policy that allows one mismatch in the alignment. A valid alignment with one mismatch falls into one of two cases according to which half of the read contains the mismatch. Bowtie proceeds in two phases corresponding to those two cases. Phase 1 load the forward index into memory and invokes the aligner with the constraint that it may not substitute at positions in the query's right half. Phase 2 uses the mirror index and invokes the aligner on the reversed query, with the constraint that the aligner may not substitute at positions in the reversed query's right half (the original query's left half). The constraints on backtracking into the right half prevent excessive backtracking, whereas the use of two phases and two indices maintains full sensitivity.

Unfortunately, it is not possible to avoid excessive backtracking fully when alignments are permitted to have two or more mismatches. Excessive backtracking is significant only when a read has many low-quality positions and does not align or aligns poorly to the reference.

4 The Three Phases of the Bowtie Algorithm for the MAQ-like Policy [9, 10]

Bowtie allows the user to select the number of mismatches

permitted (default: two) in the high-quality end of a read (default: the first 28 bases) as well as the maximum acceptable quality distance of the overall alignment (default: 70). Quality values are assumed to follow the definition in PHRED [7], where p is the probability of error and Q = -10log p. Both the read and its reverse complement are candidates for alignment to the reference. This discussion considers only the forward orientation.

The first 28 bases on the high-quality end of the read are termed the '**seed**'. The seed consists of two halves: the 14 base pair (bp) on the **high-quality end** (usually the 5' end) and the 14 bp on the **low-quality end**, termed the '**hi-half**' and the '**lo-half**', respectively. Assuming the default policy (**two mismatches permitted in the seed**), a reportable alignment will fall into one of **four cases**:

case 1: no mismatches in seed,

case 2: no mismatches in hi-half, one or two mismatches in lo-half;

case 3: no mismatches in lo-half, one or two mismatches in hi-half and

case 4: one mismatch in hi-half, one mismatch in lo-half.

All cases allow any number of mismatches in the non-seed part of the read and all cases are also subject to the quality distance constraint.

The Bowtie algorithm consists of three phases that alternate between using the forward and mirror indices, as illustrated in Figure 7. Phase 1 uses the mirror index and invokes the aligner to find alignments for cases 1 and 2. Phases 2 and 3 cooperate to find alignments for case 3: Phase 2 finds partial alignments with mismatches only in the hi-half and phase 3 attempts to extend those partial alignments into full alignments. Finally, phase 3 invokes the aligner to find alignments for case 4.

5 Full Bowtie Algorithm [1]

The full Bowtie algorithm consists of four phases. The full algorithm considers both the forward-oriented read and reverse-complement of the read. Incorporating the reverse

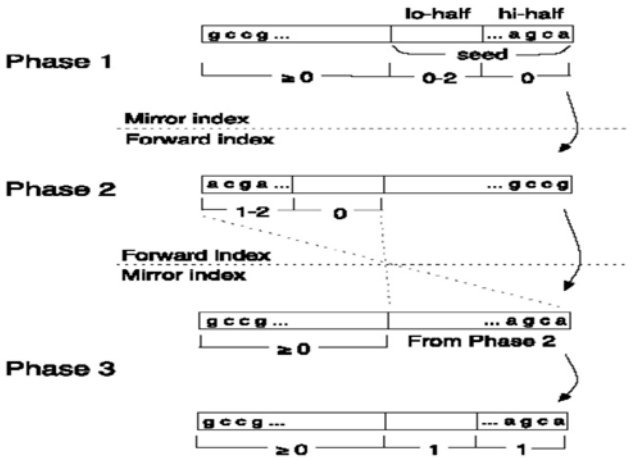

Fig. 7. The three phases of the Bowtie algorithm for the Maq-like policy. A three phase approach finds alignments for two-mismatch cases 1 to 4 while minimizing backtracking. Phase 1 uses the mirror index and invokes the aligner to find alignments for cases 1 and 2. Phases 2 and 3 cooperate to find alignments for case 3: Phase 2 finds partial alignments with mismatches only in the hi-half, and phase 3 attempts to extend those partial alignments into full alignments. Finally, phase 3 invokes the aligner to find alignments for case 4. Source: [1].

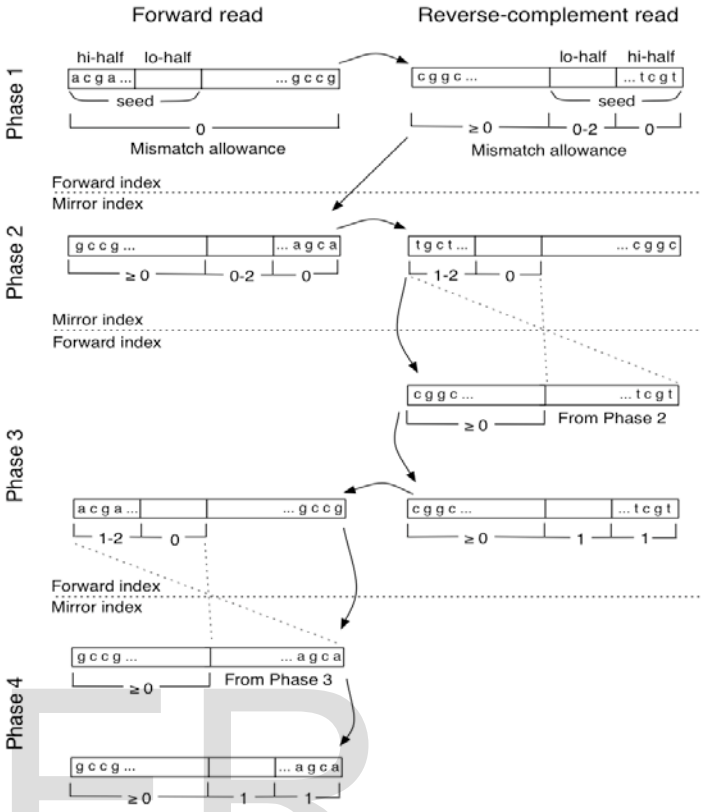

Fig. 8. The four phases of the Bowtie algorithm.

complement requires introducing a new phase to the beginning of the algorithm that uses the forward index. The new phase becomes Phase 1 and the three phases described previously become Phases 2-4. The steps required to align the reverse-complement read are analogous to those of the forward-oriented read, but shifted forward by one phase. The entire process is "packed" into four phases by interleaving the processing of the forward- oriented and reverse-complement versions of the read.

Finally, we add a check to the beginning of Phase 1 to find an end-to-end alignment with no mismatches for the forward-oriented read, if one exists. In this way, we guarantee that alignments with no mismatches will always be preferred over alignments with one or more mismatches.

6 Software

Bowtie is written in C++. Bowtie is free, open source software available from the Bowtie website [6].

7 Conclusion

In this work, we extensively analyzed different strategies for mapping of reads to a reference genome. Bowtie does not yet support paired-end alignment or alignments with insertions or deletions, although both improvements are planned for the

future. Paired-end alignment is not difficult to implement in Bowtie's framework, and we expect that Bowtie's performance advantage will be comparable to, though perhaps somewhat less than, that of unpaired alignment mode. Support for insertions and deletions are also a conceptually straightforward addition **[1].**

Acknowledgment

I express my sincere gratitude to **Professor Samar Sen Sarma** (Department of Computer Science & Engineering, **University of Calcutta**) and Assistant Professor **Sunirmal Khatua** (**University of Calcutta**). They helped me immensely by providing all sorts of reference books and articles and guided me with his valuable suggestions and timely advice. This project would have remained incomplete without their valuable feedback.

I would also like to express my gratitude to **Jadavpur University** for providing me with a suitable working environment.

REFERENCES

[1] B. Langmead, C. Trapnell, M. Pop, and S. L. Salzberg, "Ultrafast and memory-efficient alignment of short dna sequences to the human genome." Genome biology, vol. 10, no. 3, pp. R25+, 2009.

[2] Michael Burrows and David Wheeler, "A block sorting lossless data compression algorithm", Technical Report 124, Digital Equipment Corporation, 1994.

[3] P. Ferragina and G. Manzini, "Opportunistic data structures with applications," in Proc. of the 41st Annual Symposium on Foundations of Computer Science, 2000.

[4] Zhumur Ghosh, Bibekanand Mallick, "BIOINFORMATICS Principles and Applications", OXFORD UNIVERSITY PRESS, 2008.

[5] U. Manber and E. W. Myers, Suffix arrays: A new method for on-line string searches. SIAM Journal on Computing, Volume 22, No. 5, October 1993, pp.935-948.

[6] Bowtie: An ultrafast memory-efficient short read aligner [http://bowtie.cbcb.umd.edu/]

[7] Ewing B, Green P: Base-calling of automated sequencer traces using phred. II. Error robabilities. Genome Res 1998, 8:186-194.

[8] H. Li, J. Ruan, and R. Durbin, "Mapping short DNA sequencing reads and calling variants using mapping quality scores," Genome Research, vol. 18, pp. 1851–1858, 2008.

[9] R. Li, Y. Li, K. Kristiansen, and J. Wang, "SOAP: short oligonucleotide alignment program," Bioinformatics, vol. 24, no. 5, pp. 713–714, 2008.

[10] H. Lin, Z. Zhang, M. Zhang, B. Ma, and M. Li, "ZOOM! Zillions of oligos mapped," Bioinformatics, vol. 24, no. 21, pp. 2431–2437, 2008.

[11] R. A. Lippert, "Space-efficient whole genome comparisons with Burrows-Wheeler transforms," Journal of Computational Biology, vol. 12, no. 4, pp. 407–415, 2005.

[12] T. Lam, W. Sung, S. Tam, C. Wong, and S. Yiu, "Compressed indexing and local alignment of DNA," Bioinformatics, vol. 24, no. 6, pp. 791–797, 2008.

[13] D. Gusfield, Algorithms on strings, trees, and sequences. Cambridge University Press, 1997.

[14] M.-Y. Kao, Ed., Encyclopedia of Algorithms. Springer, 2008.

[15] D. Bentley, "Whole-genome re-sequencing," Curr. Opin. Genet. Dev., vol. 16, pp. 545–552, 2006.

[16] L. Hillier, G. Marth, A. R. Quinlan, and D. D. et al., "Whole-genome sequencing and variant discovery in C. elegans," Nature Methods, vol. 5, pp. 183–188, 2008.

[17] D. R. Bentley, S. Balasubramanian, and H. P. S. et al., "Accurate whole human genome sequencing using reversible terminator chemistry," Nature, vol. 456, no. 7218, pp. 53–59, 2008.

[18] J. Wang, W. Wang, and R. L. et al., "The diploid genome sequence of an Asian individual," Nature, vol. 456, no. 7218, pp. 60–65, 2008.