Efficient and Error-Tolerant Sequencing Read Mapping

Piotr Jaroszyński and Norbert Dojer

Institute of Informatics, University of Warsaw,
Banacha 2, 02-097 Warszawa, Poland
dojer@mimuw.edu.pl

Abstract. Most efficient read mappers build a *Ferragina-Manzini index* of a genome sequence and then process reads against it. In order to handle differences between reads and corresponding genome fragments, approximate read occurrences are searched in the index. This technique is particularly efficient for mapping reads of length $\sim 30bp$ with up to 2-3 errors, as first massive sequencers required.
However, within the last few years, in most popular sequencing technologies read length increased to $75 - 200bp$. Since the number of required index queries is exponential with respect to the number of errors, it is hard to maintain the allowed error rate within this method.
We propose a new approach that overcomes this problem. The main idea is to use the Ferragina-Manzini index to filter potential approximate read occurrences. Filtering is based on the *intermediate partitioning* concept, i.e. reads are split into parts, which are searched in index with reduced number of errors.
We implemented this method in *Bmap* program. Our experiments show that Bmap outperforms current methods in efficiency without sacrificing mapping accuracy.

Keywords: next generation sequencing, read mapping, Ferragina-Manzini index, intermediate partitioning

1 Introduction

Output datasets of massively parallel sequencing technologies contain sequences of millions of short DNA fragments, called *reads*. Usually, in the first step of analysis reads are *mapped* onto a reference genome, i.e. genome locations that best suit particular reads are found.

The most efficient read mappers [8, 12, 10, 7, 11, 13, 14] build a *Ferragina-Manzini index* (FM-index) of a genome sequence and then process reads against it. FM-index is a data structure based on Burrows-Wheeler transform of indexed text. It allows extremely fast and memory economical locating exact sequence occurrences in a genome. In order to locate approximate occurrences, Bowtie [8] and BWA [10] apply *neighborhood generation*, i.e. generate sequences similar to reads and search the index for them. This strategy works well when reads

are mapped with at most 1-2 errors. When more errors are allowed, efficiency rapidly drops since the neighborhood size rises exponentially.

Neighborhood searching may be sped up with *backtracking*. This technique avoids repetitive computations in searching for words sharing suffixes. The gain in computation time is proportional to the length of a common suffix. Thus, the acceleration is significant if most considered errors are located at the beginning of a read.

Bowtie creates two FM-indexes: *forward* for a genome sequence and *mirror* for the reverse one. Read neighborhood is split into two parts based on the error location and each index is searched for relevant neighborhood part. This strategy enables to increase the allowed number of errors to 3. In order to allow mapping with more errors, read *seeds* must be specified (first 28 bp by default). Then approximate seed occurrences (with ≤ 3 errors) are found with double FM-index and their extendability to whole-read alignments is verified. However, for $e > 3$, this approach does not guarantee that all read occurrences with $\leq e$ errors are identified.

BWA also creates two FM-indexes, but the mirror index serves only to the calculation of lower bounds of a minimal number of errors in genome occurrences for read prefixes. Read neighborhoods are searched with backtracking in the forward index and the bounds are used to exclude from the search space neighbors with errors aggregated at the end of the sequence. This strategy significantly reduces search time for higher error rates and, as opposite to Bowtie, BWA may identify all read occurrences with $\leq e$ errors for any e.

According to contemporary capabilities of sequencing technologies, the above tools were designed for mapping reads of length $\leq 50bp$. In this case they provide efficient mapping with satisfactory number of errors for most applications. Approaches using other indexing methods have an order of magnitude higher running time (e.g. MrsFAST [6]) and/or memory requirements (e.g. SHRIMP2 [2]).

Present-day sequencers output reads of length up to $75bp$ (5500 Series SOLiD System), $150bp$ (Illumina Genome Analyzer IIx) or even $200bp$ (Personal Genome Machine Sequencer). Longer reads entail more errors per read and, consequently, rapid rise of mapping time. To allow higher error rate, FM-index-based seed-and-extend heuristics Bowtie2 [7] and BWA-SW [11] were proposed. They save the efficiency at the cost of the certainty of finding best mapping position.

In the present work we propose *Bmap*, mapping tool efficiently finding all read occurrences with assumed number of errors. Our program is designed to reads of length $\geq 50bp$ and error rates up to 10%. Bmap uses FM-index to filter potential approximate read occurrences. Filtering is based on the *intermediate partitioning* concept [16–18], i.e. reads are split into segments, which are searched in index with reduced number of errors. Next the extendability of fund approximate segment occurrences to whole-read alignments is examined.

The idea of searching read subsequences in genome index is also used in seed-and-extend methods. However Bmap thoroughly selects seeds according to read

length and error-rate. Therefore, as opposed to heuristic approaches, it finds all read alignments with assumed number of errors.

2 Algorithm

2.1 Genome Index

The core data structure of the *Bmap* program is the index of a reference genome. It provides fast searching for exact occurrences of query sequences and extracting genome subsequences. Moreover, in order to handle efficient searching for approximate query occurrences, some additional operations dealing with partial results of exact searching should be supported. Here is the full list of operations required by our mapping algorithm:

Find$(Q) \rightarrow R$ searches for all occurrences of sequence Q and returns an opaque result R that can be used with other operations.

FindSuffixes$(Q_{1..m}) \rightarrow R_{1..m}$ works just like Find, but returns results for each suffix of Q so that R_i is the result of searching for $Q_{i..m}$.

FindContinue$(Q_{1..m}, R_{old}, f) \rightarrow R_{new}$ just like Find searches for all occurrences of $Q_{1..m}$, but takes advantage of an earlier result R_{old}, assumed to be obtained by searching for $Q_{f..m}$, and returns a new result R_{new}.

Count$(R) \rightarrow k$ returns the number of occurrences k represented by R.

Locate$(R) \rightarrow l_{1..k}$ returns locations of occurrences represented by R.

Extract$(b, l) \rightarrow S$ retrieves a subsequence of the reference sequence T: $S = T[b..b + l - 1]$.

It was shown in [4] that various variants of FM-index [3] are the most efficient data structures supporting the above operations. We chose a member of FM-index family called *Succinct Suffix Array* (SSA) [15] to be the core data structure of the *Bmap* program. The main difference between SSA and common FM-index used in Bowtie and BWA is that the latter stores explicitly Burrows-Wheeler transform of a sequence, while SSA exploits the *Wavelet Tree* structure [5]. Wavelet trees encode sequences as collections of bit vectors and take advantage of efficient bit vector navigation operations.

We performed a DNA-oriented SSA implementation, purposed to work with sequences over fixed 4-letter alphabet. We also implemented its variant, called SSAT, which explicitly stores a reference sequence. In this way, at the price of additional memory usage, relatively expensive and substantially exploited **Extract** computations are avoided. Both variants of the index have a few parameters that determine the balance between space and time requirements of the algorithm. The detailed description of implementation and parameter settings is included in Supplementary Materials. For further comparison we chose four parameter sets, presenting the range of Bmap computational requirements (see Table 2).

2.2 Intermediate Partitioning

Let $d(A, B)$ be a distance between sequences A and B, i.e. minimum number of errors required to convert A into B. The following considerations apply to both *Hamming* distance (every error is a symbol substitution) and *edit* distance (symbol insertions and deletions are also allowed). The set of all sequences B satisfying $d(A, B) \leq e$ is called e-neighborhood of A.

The following observation is a special case of a result proved in [17].

Lemma 1. *Let $A = A_1 \dots A_k$ and B be two sequences such that $d(A, B) < k \cdot j$. Then at least one subsequence A_i occurs in B with $< j$ errors.*

The above lemma justifies the following 2-phase approach to the mapping problem, called intermediate partitioning [16, 17]:

Filtration Read A is split into k segments and the genome index is searched for $(j-1)$-neighborhood of each segment.

Verification Every approximate segment occurrence is examined whether it is extendable to a whole-read alignment.

Due to the lemma, each A-occurrence with $< k \cdot j$ errors will be identified in this way.

The efficiency of intermediate partitioning depends on the balance between the time spend by the algorithm on filtration and verification phase. The filtering cost grows exponentially with j, so we decided to limit this parameter to 2. Setting $j = 1$ in the lemma above we obtain:

Corollary 1. *If A is split into $e + 1$ non-overlapping segments then in every A-occurrence with $\leq e$ errors at least one segment has an exact occurrence.*

Similarly, setting in the lemma $j = 2$ we obtain:

Corollary 2. *If A is split into $\lfloor \frac{e}{2} \rfloor + 1$ non-overlapping segments, then in every A-occurrence with $\leq e$ errors at least one segment occurs with at most one error.*

The cost of the verification phase is proportional to the number of potential read occurrences left after filtering. It depends primarily on segment length – the shorter segment is, the higher is number of its occurrences. More precisely, shortening segment length by one nucleotide quadruples the expected number. In order to have ≤ 1 occurrence per segment on average, we prefer to avoid segments shorter than 4-based logarithm of genome length.

In the case of human genome this choice results in minimum segment length $16bp$. Combining this bound with the result of Corollary 2 we obtain the limit on the allowed error number to be $2\lfloor \frac{m}{16} \rfloor - 1$, where m is the length of a read. For example, in the case of $100bp$-long reads it gives 11 errors.

2.3 Mapping Process

In order to speed up mapping low-error reads we divided the whole mapping process into three phases:

1. Mapping whole read without errors.
2. Mapping read segments without errors.
3. Mapping 1-neighborhoods of read segments.

The second and third phases may be not necessary depending upon the desired maximal number of errors. Moreover after finding a mapping that meets the error criteria it may be returned immediately. The default behavior is to return the mapping if it is certain to be the best one (i.e. has the smallest possible number of errors). In particular:

- If a mapping without errors is found, it is the best.
- If a mapping with a single error is found during the second phase, it is the best.
- If the best mapping found during the second phase has at most $k-1$ errors, it is the best one at all.
- If a mapping with at most k errors is found during the third phase, it is the best one.

The pseudo-code for the mapping process is presented as Algorithm 2.1. *MaybeReturn(errors)* indicates the points where a mapping may be returned immediately according to the above criteria.

2.4 Practical Concerns

Two characteristics of available reference genomes complicate efficient mapping: repetitiveness and incompleteness. In the present section we motivate our approach to these difficulties based on statistical analysis of the human reference genome assembled by NCBI, build 37 [1]. However, similar considerations apply to all large genomes.

Genome Repetitiveness As was shown in previous sections, *Bmap* relies on finding occurrences of relatively short read subsequences and verifying the surroundings of each occurrence. Although most subsequences have very few occurrences, there are also extremely frequent ones, due to the repetitive genome structure. For example, $> 95\%$ of $20bp$-long sequences occurring in the genome are unique, $> 99\%$ have no more than 5 occurrences, $> 99.9\%$ no more than 10. Furthermore, only 0.02% sequences occur 100-1000 times and only 0.002% over 1000 times.

There are two causes of extremely frequent sequences: low-complexity regions and repetitive elements. Table 1 illustrates both. The first group is represented by 6 most common $20bp$-long sequences: 2 of them are mono-nucleotide and next 4 sequences are built of repetitions of one bi-nucleotide pattern (there are 2

Algorithm 2.1 Map query Q by dividing into k substrings and introducing single substitutions

```
1: function MAP(Q, k, e)
2:     R ← Find(Q)                                   ▷ Try searching without errors
3:     if Count(R) > 0 then
4:         l_{1..n} = Locate(R)
5:         return l_1
6:     end if
7:     b_{1..k}, e_{1..k} ←Divide(Q, k)   ▷ Divide divides the string Q into k substrings and returns the begin and end positions for each of them
8:     S ← ∅                              ▷ Initialize a set of found approximate occurrences
9:     for i ← 1, k do                    ▷ Search for all substrings without errors
10:        R ← Find(Q[b_i..e_i])
11:        if Count(R) > 0 then
12:            Verify(Q, b_i, R, 1)
13:        end if
14:    end for
15:    MaybeReturn(k − 1)
16:    for i ← 1, k do:
17:        q ← Q[b_i..e_i]
18:        l ← |q|
19:        R_{1..l} ← FindSuffixes(q)
20:        m ← argmax_m{Count(R_m)= 0}   ▷ q_{m..l} is the shortest suffix that doesn't have any matches
21:        for p ← m, l do        ▷ Search for all substrings with a single substitution
22:            t ← q_p
23:            for all s ∈ ¬t do ▷ ¬t is the set of all symbols except the one stored in t
24:                q_p ← s                                  ▷ Introduce the substitution
25:                r ←FindContinue(q, p + 1, R_{p+1}) ▷ Search for the modified substring
26:                if Count(r) > 0 then
27:                    Verify(Q, b_i, r, k)
28:                end if
29:            end for
30:            q_p ← t                                      ▷ Restore the original value
31:        end for
32:    end for
33:    return min(S)
34: end function
35: function VERIFY(Q, R, o, mrerrs)        ▷ Query Q, result R of searching for Q_{o..}
36:    l_{1..n} ← Locate(R)                                 ▷ Get matches
37:    for i ← 1, n do
38:        errs ← d(Q,Extract(l_i − o, |Q|))                ▷ Calculate the number of errors
39:        if errs < e then
40:            S.add(⟨errs, l_i − o⟩)
41:            MaybeReturn(mrerrs)
42:        end if
43:    end for
44: end function
```

Substring	Occurrences
TTTTTTTTTTTTTTTTTTTT	451296
AAAAAAAAAAAAAAAAAAAA	447468
GTGTGTGTGTGTGTGTGTGT	246066
ACACACACACACACACACAC	243148
TGTGTGTGTGTGTGTGTGTG	241608
CACACACACACACACACACA	238826
CTCCCAAAGTGCTGGGATTA	170026
TAATCCCAGCACTTTGGGAG	169758
CCTCCCAAAGTGCTGGGATT	166855
AATCCCAGCACTTTGGGAGG	166726

Table 1. Most frequent genome subsequences of length 20

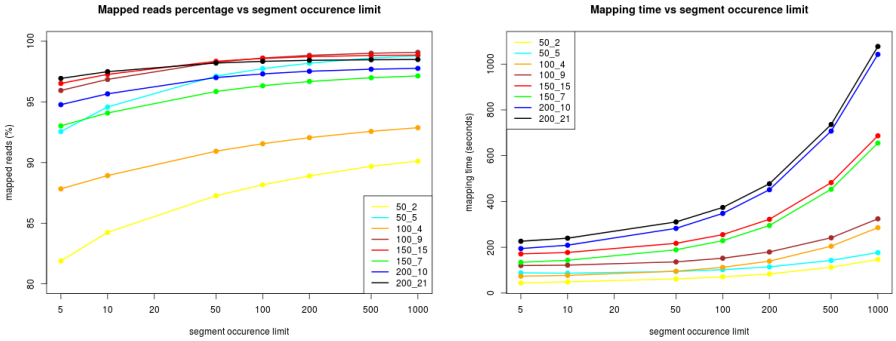

Fig. 1. Percentage of mapped reads and mapping time vs limit on the number of segment occurrences.

pairs of complementary sequences). The remaining 4 sequences origin from both strands of a 21*bp*-long fragment of the Alu repetitive element.

Frequently occurring read segments drastically lengthen the verification phase. Moreover, if all the segments of a read have plenty of occurrences, then most likely the whole read occurs multiple times. In this case, the profit of mapping the read is usually poor, because uncertain localizations are rather useless for further analysis. Therefore we limit the number of verifications for a single segment. This approach does not obstruct mapping reads with several frequent segments – they are still mappable unless all their rare segments are highly erroneous.

We set the default limiting number of segment occurrences to 100. It means that only 0.06% of 16*bp*-long segments occurring in the human genome overdraws the limit and this percentage declines for longer segments. Results presented in Figure 1 show that our choice guarantees pretty fast mapping without sacrificing quality.

Genome Incompleteness The total length of unknown fragments in human reference genome accounts for about 8% of the whole genome. Figure 2 shows that vast majority of them is concentrated in very long contiguous series. Direct incorporation of ambiguous nucleotides into FM-index (i.e extending the sequence alphabet with a fifth symbol) would complicate the algorithms and downgrade its performance. In BWA this problem is overcome by replacing unknown fragments with random nucleotides. Authors argue that the insignificant probability of mapping a read into a random sequence legitimates this method. Our approach is similar, but we filled unknown fragments with a mono-nucleotide (G) sequence instead of a random one. In this way potential fake mappings may be avoided by simple read preprocessing.

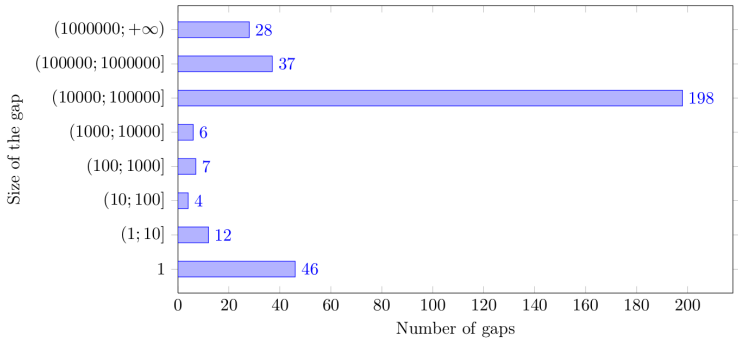

Fig. 2. Unknown parts (gaps) in the genome

3 Results

Benchmarking mapping programs is a difficult task for a few reasons. First, there are different variants of the mapping problem. Most programs try to find the best read occurrence, but some of them have another objective, e.g. MrsFAST locate all read occurrences with an assumed number of errors. Second, there are many possible evaluation objectives, e.g. mapping quality, memory usage, speed. In a fair comparison, when one of them is concerned, the rest should be adjusted similarly in all evaluated programs. Furthermore, different evaluation objectives entail their specific problems. For example, it is hard to compare the efficiency of programs for CPU- and GPU-based computations. Similarly, evaluation of mapping quality highly depends on how does benchmarking data and a quality score fit the mapper's model of sequence similarity.

We decided to focus on efficiency objectives. Consequently, we restricted our attention to FM-index-based mappers (Bowtie, BWA, Bowtie2 and BWA-SW), since other approaches are an order of magnitude less efficient. Moreover, we

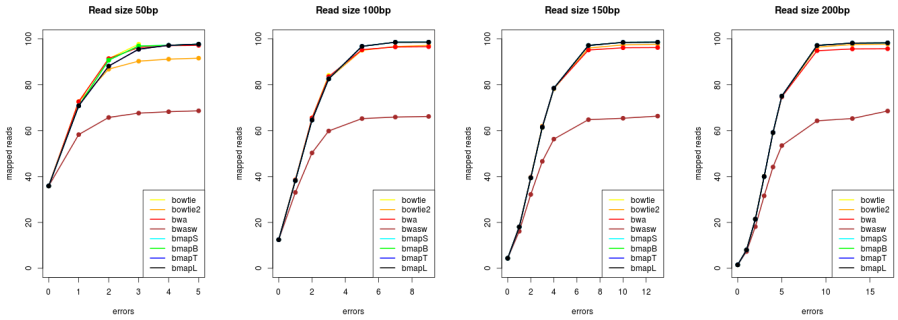

Fig. 3. Percentage of mapped reads vs number of allowed errors for reads of size $50 - 200bp$.

excluded mappers for GPU-based computations: SOAP3 [13] and CUSHAW [14] due to inability of fair efficiency comparison.

We established clear and strict quality criteria in program parameter settings: for every read, its genome occurrence with minimal number of mismatches should be returned unless this number exceeds an assumed threshold. Namely, Bowtie was run with options `--best -v ERRORS`, where `ERRORS` is the maximum allowed number of errors. We ran BWA with `-o 0 -R 1 -e ERRORS` and Bowtie2 with `--ignore-quals --mp 1,1 --rfg 100,0 --rdg 100,0 --score-min C,-ERRORS`. Finally, in BWA-SW we set the options `-b 0 -g 100 -w 1 -T MINSCORE`, where `MINSCORE` is the difference between read length and the maximum number of errors.

Assumed parameter setting guarantees similar level of specificity for all benchmarked programs. Consequently, sensitivity of mapping is determined by the percentage of mapped reads. As Figure 3 shows, the level of mappable reads is rather similar for all programs expect BWA-SW (and Bowtie2 in the case of $50bp$-long reads). Only for reads longer than $50bp$ and higher error rates Bmap performs slightly better than competitors (i.e BWA and Bowtie2, since Bowtie maps reads with ≤ 3 mismatches).

Memory usage of all programs is reported in Table 2. Bmap may be customized to fit the available memory. Its requirements are similar to those of Bowtie and smaller than those of BWA and Bowtie2. BWA-SW requires much more memory than all other programs.

Mapping speed tests were run on a server with two Intel Xeon E31245 CPUs and 16GB of RAM. Human genome assembled by NCBI, build 37 [1] was used as a reference. Synthetic reads were generated using a **wgsim** tool version 0.3.0 [9] with default parameters. It takes a genome as an input and randomly chooses positions from which reads are generated. Moreover, errors and mutations are introduced to reads.

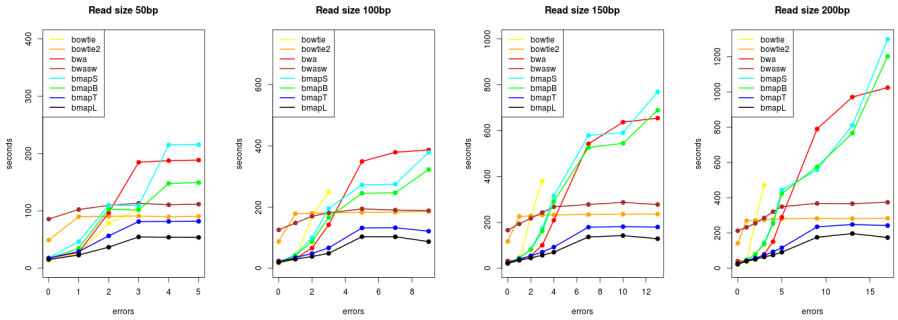

Fig. 4. Mapping time vs number of allowed errors for reads of size $50 - 200bp$.

The computational time of mapping is reported in Figure 4. Both SSA-based Bmap versions ar as fast as Bowtie and BWA. BmapS is not much slower than BmapB and can be a good solution for low memory environments. Bowtie2 and BWA-SW are relatively slow for low error rates, but their speed is almost independent of the number of errors. Consequently, for long sequences and very high error rates they are $3 - 4$ times faster than BWA, BmapS and BmapB. BmapT and BmapL are the fastest in all cases.

4 Conclusion

We proposed Bmap, a new algorithm for mapping reads from next generation sequencing. Similarly to Bowtie and BWA, Bmap is designed to find all read occurrences with assumed maximum number of errors. Our program has highly customizable balance between memory requirements and computation time: Bmap with SSA index maps reads as soon as Bowtie and BWA using less memory; Bmap with SSAT index works much faster than competitors with similar memory requirements. The speed-up is most evident for higher error rates, because

Name	Index type	Index size for human genome
BmapS	SSA	1.8GB
BmapB	SSA	2.4GB
BmapT	SSAT	2.4GB
BmapL	SSAT	2.9GB
Bowtie	FM-index	2.3GB
BWA	FM-index	3.1GB
Bowtie2	FM-index	3.2GB
BWA-SW	FM-index	5.2GB

Table 2. Memory requirements of compared mapping programs. BmapX refers to Bmap variants.

intermediate partitioning strategy of Bmap is better suited for this case than neighborhood generation with backtracking used in the other tools.

Two seed-and-extend heuristics: Bowtie2 and BWA-SW were included into our comparison. These tools are also designed to map long reads with high error rate, but their preferred model of sequence similarity is different than the one considered in this study. Therefore the reduction of their mapped reads percentage, especially visible for BWA-SW, is probably a side effect of consistent parameter setting for our benchmark. On the other hand, computation time of Bowtie2 and BWA-SW is almost independent of the number of errors, so these tools are very fast when mapping long reads with high error rate. However, even for such data Bmap with SSAT-based index works over 1.5× faster and occupies less memory space.

Acknowledgements This research was supported by NCN grant 2011/01/B/NZ2/00864.

Additional Data Program code and Supplementary Materials are available at `http://bioputer.mimuw.edu.pl/papers/bmap/`.

References

1. 1000-Genomes-g1k_v37: Sequenced human genome g1k_v37 (2010), ftp://ftp.1000genomes.ebi.ac.uk/vol1/ftp/technical/reference/human_g1k_v37.fasta.gz
2. David, M., Dzamba, M., Lister, D., Ilie, L., Brudno, M.: Shrimp2: sensitive yet practical short read mapping. Bioinformatics 27(7), 1011–1012 (Apr 2011), http://dx.doi.org/10.1093/bioinformatics/btr046
3. Ferragina, P., Manzini, G.: Opportunistic data structures with applications. In: Proceedings of the 41st Annual Symposium on Foundations of Computer Science. pp. 390–. IEEE Computer Society, Washington, DC, USA (2000), http://dl.acm.org/citation.cfm?id=795666.796543
4. Ferragina, P., González, R., Navarro, G., Venturini, R.: Compressed text indexes: From theory to practice! CoRR abs/0712.3360 (2007)
5. Grossi, R., Gupta, A., Vitter, J.S.: High-order entropy-compressed text indexes. In: Proceedings of the fourteenth annual ACM-SIAM symposium on Discrete algorithms. pp. 841–850. SODA '03, Society for Industrial and Applied Mathematics, Philadelphia, PA, USA (2003), http://dl.acm.org/citation.cfm?id=644108.644250
6. Hach, F., Hormozdiari, F., Alkan, C., Hormozdiari, F., Birol, I., Eichler, E.E., Sahinalp, S.C.: mrsfast: a cache-oblivious algorithm for short-read mapping. Nat Methods 7(8), 576–577 (Aug 2010), http://dx.doi.org/10.1038/nmeth0810-576
7. Langmead, B., Salzberg, S.L.: Fast gapped-read alignment with bowtie 2. Nat Methods 9(4), 357–359 (Apr 2012), http://dx.doi.org/10.1038/nmeth.1923
8. Langmead, B., Trapnell, C., Pop, M., Salzberg, S.: Ultrafast and memory-efficient alignment of short dna sequences to the human genome. Genome Biology 10(3), R25 (2009), http://genomebiology.com/2009/10/3/R25
9. Li, H.: wgsim version 0.3.0 - reads simulator (2011), https://github.com/lh3/wgsim

10. Li, H., Durbin, R.: Fast and accurate short read alignment with burrows–wheeler transform. Bioinformatics 25(14), 1754–1760 (2009), http://bioinformatics.oxfordjournals.org/content/25/14/1754.abstract
11. Li, H., Durbin, R.: Fast and accurate long-read alignment with burrows-wheeler transform. Bioinformatics 26(5), 589–595 (Mar 2010), http://dx.doi.org/10.1093/bioinformatics/btp698
12. Li, R., Yu, C., Li, Y., Lam, T.W., Yiu, S.M., Kristiansen, K., Wang, J.: Soap2: an improved ultrafast tool for short read alignment. Bioinformatics 25(15), 1966–1967 (Aug 2009), http://dx.doi.org/10.1093/bioinformatics/btp336
13. Liu, C.M., Wong, T., Wu, E., Luo, R., Yiu, S.M., Li, Y., Wang, B., Yu, C., Chu, X., Zhao, K., Li, R., Lam, T.W.: Soap3: ultra-fast gpu-based parallel alignment tool for short reads. Bioinformatics 28(6), 878–879 (Mar 2012), http://dx.doi.org/10.1093/bioinformatics/bts061
14. Liu, Y., Schmidt, B., Maskell, D.L.: Cushaw: a cuda compatible short read aligner to large genomes based on the burrows-wheeler transform. Bioinformatics 28(14), 1830–1837 (Jul 2012), http://dx.doi.org/10.1093/bioinformatics/bts276
15. Mäkinen, V., Navarro, G.: Succinct suffix arrays based on run-length encoding. Nordic Journal of Computing 12(1), 40–66 (2005)
16. Myers, E.: A sublinear algorithm for approximate keyword searching. Algorithmica 12, 345–374 (1994), http://dx.doi.org/10.1007/BF01185432, 10.1007/BF01185432
17. Navarro, G., Baeza-Yates, R.: A hybrid indexing method for approximate string matching. Journal of Discrete Algorithms (JDA) 1(1), 205–239 (2000), special issue on Matching Patterns.
18. Navarro, G., Baeza-Yates, R., Sutinen, E., Tarhio, J.: Indexing methods for approximate string matching. IEEE Data Engineering Bulletin 24, 2001 (2000)