BIOINFORMATICS ORIGINAL PAPER

Vol. 28 no. 8 2012, pages 1070–1077
doi:10.1093/bioinformatics/bts102

Sequence analysis

Advance Access publication March 7, 2012

YOABS: yet other aligner of biological sequences—an efficient linearly scaling nucleotide aligner

V. L. Galinsky[1,2]

[1]Qualg Inc., La Jolla, CA, 92037, USA and [2]ECE Department, UCSD, 9500 Gilman Dr, MC #407, La Jolla, CA, 92093-0407

Associate Editor: John Quackenbush

ABSTRACT

Motivation: Explosive growth of short-read sequencing technologies in the recent years resulted in rapid development of many new alignment algorithms and programs. But most of them are not efficient or not applicable for reads $\gtrsim$200 bp because these algorithms specifically designed to process short queries with relatively low sequencing error rates. However, the current trend to increase reliability of detection of structural variations in assembled genomes as well as to facilitate *de novo* sequencing demand complimenting high-throughput short-read platforms with long-read mapping. Thus, algorithms and programs for efficient mapping of longer reads are becoming crucial. However, the choice of long-read aligners effective in terms of both performance and memory are limited and includes only handful of hash table (BLAT, SSAHA2) or trie (Burrows-Wheeler Transform - Smith-Waterman (BWT-SW), Burrows-Wheeler Alignerr - Smith-Waterman (BWA-SW)) based algorithms.

Results: New $O(n)$ algorithm that combines the advantages of both hash and trie-based methods has been designed to effectively align long biological sequences ($\gtrsim$200 bp) against a large sequence database with small memory footprint (e.g. $\sim$2 GB for the human genome). The algorithm is accurate and significantly more fast than BLAT or BWT-SW, but similar to BWT-SW it can find all local alignments. It is as accurate as SSAHA2 or BWA-SW, but uses 3+ times less memory and 10+ times faster than SSAHA2, several times faster than BWA-SW with low error rates and almost two times less memory.

Availability and implementation: The prototype implementation of the algorithm will be available upon request for non-commercial use in academia (local hit table binary and indices are at ftp://styx.ucsd.edu).

Contact: vit@ucsd.edu

Supplementary information: Supplementary data are available at *Bioinformatics* online.

Received on August 3, 2011; revised on January 5, 2012; accepted on February 24, 2012

1 INTRODUCTION

Development of sensitive local alignment algorithms was started in late 1980 with several pioneering tools such as FASTA (Pearson and Lipman, 1988) and BLAST (Altschul *et al.*, 1997). For alignments of highly similar sequences to genomes, they were followed later by a new generation of faster methods, e.g. MegaBLAST (Morgulis *et al.*, 2008; Zhang *et al.*, 2000), SSAHA2 (Ning *et al.*, 2001), BLAT (Kent, 2002) and PatternHunter (Ma *et al.*, 2002). Next-generation sequencing technologies pushed development of new algorithms even further for efficient processing of millions of short ($\lesssim$100 bp) reads. These ultra-fast tools were orders of magnitude faster and included SOAP (Li *et al.*, 2009b), MAQ (Li *et al.*, 2008), Bowtie (Langmead *et al.*, 2009), BWA (Li and Durbin, 2009), Stampy (Lunter and Goodson, 2011), etc. However, emerging single molecule sequencing technologies are constantly pushing read lengths into longer and longer realm ($\gtrsim$1K base pairs and more). Most of these ultra-fast short read tools do not perform well on these not so short reads as the tools were exclusively designed for reads $\sim$100 bp. But efficiently aligning long reads ($\gtrsim$200 bp) against a long reference sequence ($\gtrsim$1 Gb, like e.g. the human genome) has different overall objectives and hence represents a different challenge to the development of alignment tools.

In contrast to short-read alignment when the best match is deduced by the end-to-end mapping of the query to the reference that minimizes a number of mismatches, long-read alignment is often based on several local matches and thus is being able to detect both structural variations in the query and erroneous assemblies in the reference. Additionally, short-read aligners are optimized for ungapped alignment and introduction of even limited number of short (several base pairs) gaps impose heavy performance penalties on these short-read algorithms. Long-read aligners on the contrary should be able (and optimized) to deal with arbitrary number of gaps of arbitrary size each.

The majority of currently available long-read alignment algorithms may be classified as either using hash table indexing, like in BLAT (Kent, 2002) or in SSAHA2 (Ning *et al.*, 2001), or using some sort of compressed trie indexing based on Burrows–Wheeler transform (BWT) (Burrows and Wheeler, 1994), for example, in BWT-SW (Lam *et al.*, 2008) or in BWA-SW (Li and Durbin, 2010). But in spite of using different indexing strategies all the above long alignment algorithms follow the seed-and-extend paradigm, i.e. they first search for one or more of the so called seeds (either short exact matches, as in SSAHA2 and BLAT, or longer gapped matches in unique regions, as in BWA-SW). The found seeds are then extended to cover the whole query sequence using the Smith–Waterman algorithm (BWA-SW uses this algorithm for identifying long gapped seeds as well). This extension algorithm is computationally expensive and although long-read aligners introduce various heuristic accelerations to limit use of this computationally penalizing phase by reducing unnecessary seed extensions especially in highly repetitive regions, the resulting toll

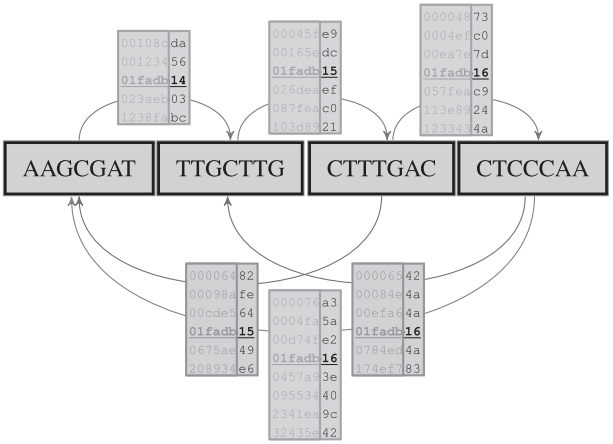

Fig. 1. A schematic example of building forward (top) and backward (bottom) indices for the reference sequence. The input sequence is partitioned into a series of l bp prefixes and m bp suffixes and arrows are used to indicate all the prefix–suffix pairs. The position in the reference where this particular prefix–suffix combination is encountered is recorded in a table connected to the arrow that indicates this prefix–suffix pair. The shaded part of the tables indicates masking of each location with k-bit mask. The underlined entry in each table corresponds to the current location in the reference.

remains heavy. Even in currently fastest heuristically accelerated BWT-based algorithms (e.g. BWA-SW) this toll is indirectly present through a large constant associated with each BWT operation.

This article outlines a new long alignment algorithm, yet other aligner of biological sequences (YOABS), that does not use the seed-and-extend paradigm and, hence, does not bear those computational expenses imposed by the Smith–Waterman algorithm (As a matter of fact, it does not use the dynamic programming at all). The algorithm is designed to combine the advantages of both hash- and trie-based algorithms. Similar to hash-based algorithms it uses a series of look ups to build a correspondence between a query and a reference and, hence, has low-associated computational overhead. Similar to compressed trie algorithms it compresses highly repetitive regions and suppresses short repetitive matches that poison the performance of hash-based algorithms. The high accuracy, the low computational complexity as well as low memory requirements make the algorithm a candidate for specialty implementations with graphics processing units (GPUs) or field-programmable gate arrays (FPGAs).

The article also presents some preliminary evaluation of algorithm's possible practical performance using work-in-progress test implementation along with BWA-SW and SSAHA2 on both simulated and real data.

2 METHODS

2.1 Reference index

YOABS starts with building several forward and backward indices for the reference sequence. The indices are organized in list type structures to combine the advantages of both hash-based (BLAT, SSAHA2) and trie-based (BWT-SW, BWA-SW) algorithms. In Figure 1, the schematic example of single step of index building is shown. The forward index (shown at the top part of Fig. 1) is organized as a lexicographically sorted array of l bp prefixes. Each prefix entry is pointing to a lexicographically sorted array of m bp suffixes. In turn, each suffix entry is associated with a numerically sorted array of l scaled k-bit masked locations (i.e. locations/l modulo 2^k) of each of these $l+m$ bp indexed entries. An optimal choice of l, m and n parameters probably depends on a size and a composition of reference sequence, but for human genome with ~3 billions bp the parameters $l=m=7$ bp and $k=8$ bit seem to work well and were adopted as a feasibility checkpoint in all the results reported below using test implementation of this article. The size of this index for the human genome is roughly 3 billions bp / 7 bp × 8 bit ≈ 400 MB.

Table 1. A list of notations used in algorithms and figures

Name	Size	Description
l		Prefix length (bp)
m		Suffix length (bp)
k		Distance mask size (bit)
RefSeq	$0:$len	Reference sequence
Query	$0:q$	Query sequence
RefInd	Variable	Reference index
HitTable	$[0:l-1]\times[0:2^k-1]$	Local hit table

It looks natural to build similar index for backward direction as well but doing this will actually be bad choice as the combination of these indices will perform poorly, for example several errors clamped in the middle sections of the sequence shown in Figure 1 will neither produce any hits with the forward index nor with the backward one. The better choice would be to build two backward indices with gaps: one with $l=7$ bp gaps and the second with $2l=14$ bp gaps (as shown at the bottom of Fig. 1). As a result, these three indices (one forward and two backward) applied in concert will produce at least one hit for any $4\times l=28$ subsequence with 2 or less single base errors, that is with uniform error density of ≈7% or less.

The purpose of splitting $l+m$ bp indices into l-bp prefix and m-bp suffix is not just for conserving memory. Both the prefix and the suffix parts are lexicographically sorted, therefore, they provide not only an index (or hash), but also can work as a forward or backward tree, thus allow fast unwinding of low complexity/low error rate regions (as described in Section 2.2).

An adaptation of $k=8$ bit mask to store the locations naturally creates compressed index, as many highly repetitive parts of the reference will be collapsed into a single index entry with the same prefix, suffix and masked location.

Algorithm 1: Building reference index

input : A reference sequence RefSeq
output: A reference index RefInd

for $i \leftarrow 0$ **to** len $- l-m$ **step** l **do**
  Prefix ← 2bitsEncode(RefSeq $[i:i+l]$);
  Suffix ← 2bitsEncode(RefSeq $[i+l:i+l+m]$);
  Distance ← (i/l) modulo 2^k;
  `PushBack` Suffix in RefInd [Prefix];
  `PushBack` Distance in RefInd [Prefix, Suffix];
foreach Prefix p in RefInd **do**
  `Sort` suffixes in RefInd $[p]$;
  foreach Suffix s in RefInd $[p]$ **do**
    `Sort` distances in RefInd $[p,s]$;

A pseudo-code of the index building step is sketched as Algorithm 1. (A list of notations used in algorithms is summarized in Table 1). Outlined so far indices will allow to record hits between the query and the reference, but would not give the exact location of those hits, therefore, some additional data structures are needed to store any information required to decode the complete location, and hence to decompress (or to resolve collisions, i.e. to distinguish between different locations collapsed into the same 2^k-masked entry) in the highly repetitive regions of the index. For reported reference implementation it resulted in total addition of ≈500MB of data and will be described in Appendix.

2.2 Query sequence

Using indices described in the previous section the process of finding alignments between the query and the reference sequences starts with recording all local hits between them. The hits are organized in a table by a distance in a query (modulo l) versus a difference in distances in a reference and in a query (scaled by l and modulo 2^k). Again size of this table depends on the choice of parameters l, m and k and can be expressed as $l \times 2^k$, that is 7×256 for the case reported here. (Actually the algorithm fills up two of these tables, the second table is for the reverse compliment query sequence.)

Algorithm 2: Filling local hit table

input : A reference index RefInd
input : A query sequence Query
output: A local hit table HitTable

for $i \leftarrow 0$ **to** $q-l-m$ **do**
  Prefix $\leftarrow$ 2bitsEncode(Query $[i:i+l]$);
  Suffix $\leftarrow$ 2bitsEncode(Query $[i+l:i+l+m]$);
  foreach Distance d *in* RefInd [Prefix, Suffix] **do**
    HitTable $\left[i \text{ modulo } l, (d-i/l) \text{ modulo } 2^k\right]$++;

A pseudo-code of the local hit table building step is sketched as Algorithm 2. The time complexity of the algorithm is $O(2^k q)$, where q is the length of the query, i.e. for $k=8$ it is $O(256q)$. An illustrative example that outlines the first and the last steps of filling typical hit table is shown in Figure 2 for human reference genome and $q=84$ bp query sequence. The hit table includes not only consecutive $l+m$-mer matches (14-mer in this case), but matches with l and $2l$ gaps as well. Therefore, the perfect match at the first step (in this case first 28 bp subsequence) should result in the total number of hits not <6 plus at least 3 new hits for each next perfectly matched l bp segment. The six hits for the perfect match of the first $4l$ bp segment could be understood from the arrows in the querylet (Fig. 1), with three hits from the forward index, two hits from the backward index with l bp gap and one hit from the backward index with $2l$ bp gap. Extending the first $4l$ bp subsequence by the next l bp segment will produce one hit for each of the three indices, that is three new hits. The number of possible l bp extensions for a q bp query is $(q-q \bmod l)/l-4$, where subtraction of the number 4 takes into account the first $4l$ subsequence. Therefore, an expression for the number of hits of perfectly matched q bp query is

$$\text{Number of hits}(q) = 6+3\cdot((q-q \bmod l)/l-4)$$

Hence, for the perfect match of $q=84$ bp query sequence the hit table will contain at the last step at least one entry with the number

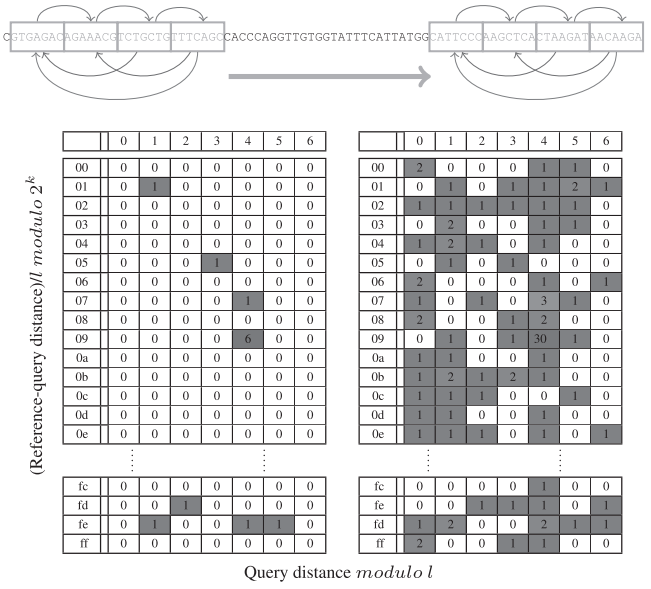

Fig. 2. A schematic example of filling a table of local hits between a query sequence and a reference index. (Different shades of gray (or colors in online version) denote different numbers of hits recorded at given table location to emphasize and distinguish between the maximums and the noise.) The top panel schematically illustrates that the query is being processed sequentially from left to right with two querylet snapshots corresponding to two states in the local hit table. The left table shows a small number of hits recorded at different values of 2^k-masked position as the querylet moved through the first l bases of the query. Several random hits were recorded with at least one prefix–suffix pair (shown by the arrows in the querylet) at 2, 3, 4, 5 and 6 position in the query and one perfect match hit was recorded at the cell (9,4). The perfect match means that the same 2^k-masked position (row 09 in this example) has been recorded for all six combinations of prefix–suffix in querylet (shown by arrows). The right hit table shows the final state as the querylet reached the end of the query (The table entries produced by hits with reverse compliment of the query sequence are not shown for simplicity).

of hits equals to 30, as example in Figure 2 shows. Presence of errors or SNPs in a query sequence will decrease this number of hits, but on the contrary multiple hits at the same 2^k-masked locations especially for highly repetitive subsequences will increase this number.

The entries with highest number of hits will be candidates for the best alignment of the query to the reference. Presence of single well separated maximum in the hit table clearly indicates the unique alignment. For a set of chosen parameters ($l=m=7$ and $k=8$) a difference of 4 or 5 between max and second max entries in the hit table is good enough to rule out random hits that may arise due to short period of the l-scaled 2^k-masked location for $k=8$.

A construction of full local hit table for the query sequence creates convenient and straightforward way of searching for significant alignments with small but arbitrary gaps as well as for detection of chimeric reads. An example in Figure 3 shows how the content of the table for the same 84 bp query sequence will be modified if a small 5 bp deletion is introduced. In this case, the leading and the trailing portions of the sequence roughly of 40 bp each will produce two maximums with $3\cdot((40-40 \bmod 7)/7-4)+6=9$ hits. Those maximums are located at cells (9,4) and $(a,6)$, that is they separated by four consecutive cells without hits or with low random hit counts [cells (9,3), (9,2), (9,1) and (9,0)]. In general, n bases

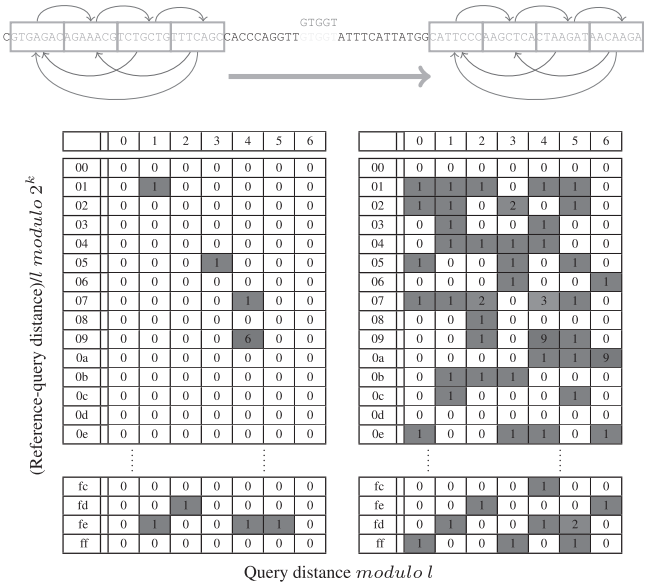

Fig. 3. A schematic example of filling a local hit table with gaps in query sequence. The left table that corresponds to the first several steps of processing the query before the gap is the same as the table in Figure 2. The right table that corresponds to the final stage includes all hits produced by the query section after the gap as well. The single maximum that was in cell $(9,5)$ in Figure 2 now occupies two cells. The first is the same $(9,4)$ and the second corresponds to the trailing portion of the query (the portion after the gap) and is located in the cell $(9+(\text{gap size}-1+l-4)/l, (4-\text{gap size}+l) \text{ modulo } l)$, that is they are separated by gap size less one cells with random hits.

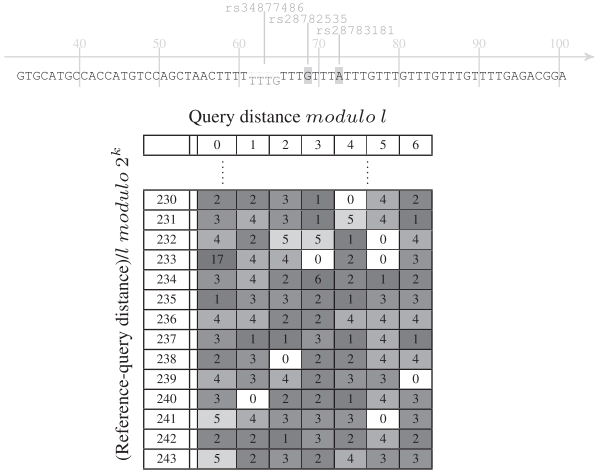

Fig. 4. A part of local hit table when both gaps and SNPs may be present in query sequence (101 bp single end Illumina read from Chromosome 1). The leading 65 bp part of the query produces a maximum in cell $(233,0)$. The trailing 36 bp part located after 4 bp deletion *rs*34877486 produces three 'empty' cells $(234,\{6\text{-}4\})$ in right–left top–down direction and increases number of hits in cell $(234,3)$, although this increase is not very significant as the trailing part includes highly repetitive subsequences TTTG.

deletion produces $n-1$ empty cells in the right–left and then top–down direction with respect to the original cell. On the contrary, n bases insertion does it in reverse, it adds $n-1$ empty/low noise cells in the left–right down–top direction.

Thus, local hit table provides easy way of finding the total size of gap or gaps between local subalignments as well as types of these gaps (insertion or deletion) by analyzing the maximums that were formed at different 2^k-masked locations. For chimeric read the hit table will contain several maximums as well but the separation between them both by the 2^k-masked location and by the full distance may in general be arbitrary large.

The presence in the query sequence highly repetitive areas (like for example multiple TTTG subsequences at the trailing part of real 101 bp single end Illumina read presumably from Chromosome 1 shown in Fig. 4) complicates the picture, especially when combined with errors, SNPs or indels. But even in this case signatures of both relatively unique ~65 bp leading part [maximum in cell (233,0)] and highly repetitive ~36 bp trailing part [maximum in cell (234,3)] separated by 4 bp deletion [cells (234,{6,5,4}) in right–left top–down direction] can be spotted from the hit table relatively easy as input from repetitive regions tends to spread more or less uniformly across many hit table entries. This 101 bp example is chosen as a stress test case and longer reads provide more clearly distinguishable entries in the hit table.

A simple greedy search algorithm has been implemented for selection of top hits from the local hit table. The algorithm first converts the 2D local hit table into linear buffer (circularly connected) following the right–left top–down pattern for the forward matches and the left–right top–down pattern for the reverse matches. The linear buffer entries is then partitioned in two classes, one corresponding to the possible hits and the second to the random background hits. Several rules are used for partitioning, the most important two rules are: three sigma rule (i.e. the entry is assumed to be a possible hit candidate if it is separated by >3 sigma from the mean) and the maximum change rule (i.e. the entry is classified as a hit candidate if it has number of hits larger than the entry with the biggest change in the sorted linear buffer). The set of hit candidates is then processed in greedy order, selecting the entry with largest number of hits, decoding the full reference position (see Algorithm 2 in Appendix) using any three consecutive perfect matches from the list, unwinding low error regions (one or two base errors) using the reference index as a tree and scavenging the neighboring maxima in the local hit table that have correct encoded positions and hence may correspond to insertions or deletions. The process stops when a single entry is formed that clearly exceeds all other entries as well as the remaining hit candidates. The pseudo-code of the algorithm is sketched as Algorithm 3 (it should be noted that several simplifications are used in the sketch, e.g. the pseudo-code does not show resolving of collisions, i.e. multiple 2^k-masked locations, see Appendix).

2.3 Final alignment

One of the most important difference of presented algorithm from the majority of long read alignment methods lies in the dismissal of seed-and-extend paradigm that is routinely employed by the most of current long read aligners. The existing algorithms (both hash and trie based) first search for the alignment seeds, but limit their number or size and keep several non-overlapped 11–12 bp fixed size seeds (BLAT, SSAHA2) or one 25–35 bp variable size (possibly with gaps) seed (BWA-SW) as candidates for further processing. Their next step involves extending seed alignments to the rest of the query sequence using the dynamic programming approaches,

Algorithm 3: Hit table processing

```
input  : A local hit table HitTable
input  : A reference index RefInd
input  : A query sequence Query
output: A hits vector Hits
for i ← 0 to 2^k − 1 do
    for j ← 0 to l − 1 do
        k ← i·l + (l − 1 − j);
        HitOneD [k] ← HitTable [j][i];
HitCandidates ← Sort (Partition (HitOneD));
foreach Hit h_0 in HitCandidates do
    q_0,EncPos [0] ← FindThreePerfectMatches(h_0);
    for i ← 1 to 3 do
        j ← q_0 + (i − 1)·l;
        EncPos [i] ← GetEncodedPos (j);
    RefPos ← DecodePos (EncPos)-q_0;
    j ← q_0 + 3l;
    while j < q do
        if GetEncodedPos (j) is in EncPos then j ← j + l
        else
            // In the order of increasing
               distance going left and right
            foreach neighbor Hit h_1 of h_0 do
                j ← value for h_1;
                if GetEncodedPos (j) is in EncPos then
                    Remove h_1 from HitCandidates ;
                    q_0,EncPos, RefPos ← values for h_1;
                    break;
    j ← q_0 − l;
    while j ≥ 0 do
        if GetEncodedPos (j) is in EncPos then j ← j − l
        else
            // Loop through h_0 neighbors (see
               above)
    Hits ← HitCandidates ;
```

usually the Smith–Waterman algorithm. Giving the remaining length of the query q and the size of the part of the reference used for alignment $r=q+g$ (allowing for possible gap or gaps of the total length g) the typical time complexity of this dynamic programming step is $O(qr)=O(q^2+gq)$.

Instead of extending relatively short seed area to the entire query sequence with this expensive (quadratic in time) dynamic programming step, the presented algorithm chooses to apply different strategy, which scales linearly $O(q)$ with the size of the query q.

When the entire query sequence has been processed and candidate entries from the hit table for the final alignment have been identified the last processing step involves decoding the unmasked location of the hits in the reference (may be resolving collisions due to finite period of 2^k-masked location if present in the hits) and glueing those hits together.

Because of the presence of single or multiple base errors, SNPs or indels there will be areas in the query sequence where no local hits are detected, thus the glueing stage should provide a way of filling all these holes. When the hole is surrounded by the local hits with

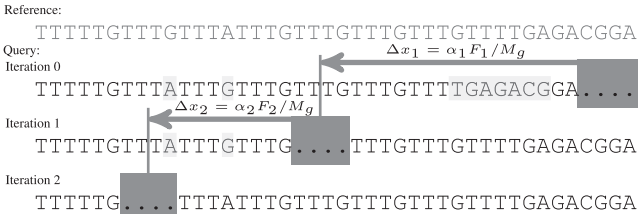

Fig. 5. A simplistic illustration of simulated annealing type $O(n)$ procedure used as an efficient alternative to $O(n^2+gn)$ Smith–Waterman algorithm. The procedure uses gap size g and type (deletion) obtained from the local hit table and finds the gap position iteratively using some cooling schedule (α_i), gap/mismatch interaction potential/force (F_i) and virtual gap mass (M_g).

the same difference between the query and the reference coordinates the filling is trivial and involves simple $O(n)$ (where n is the length of the hole, $n \ll q$) transition through all bases in the hole in order to find and record all mismatches. Theoretically it may be possible that adding equal number of equal length insertions and deletions may lower the total score, practically it is highly unlikely, giving the relatively high gap penalty used in most current algorithms and small size of the holes, especially in resequencing as well as in long read *de novo* assembly projects.

The major advantage of the algorithm emerges when the hole is located in the area between local hits with different values of the query and the reference coordinate differences (as shown in examples in Figs 3 and 4). This indicates that gap (either insertion or deletion) should be introduced somewhere in the hole to allow the transition from the leftmost to the rightmost hits. As it was shown above the local hit table allows to deduce the total size as as well as the type of the gap, but the exact position of this gap (or gaps) in the query sequence relative to the reference still needs to be determined.

It would be tempting to simply align the part of the query without local hits using the Smith–Waterman algorithm, assuming relatively short size $n \ll q$ of these holes, but the end result will have quadratic $O(n(n+g))$ complexity and may still hurt the overall performance when many of these small holes are present or when their size has increased. Besides, using quadratic algorithm would be an overkill and a complete waste of resources as more optimal linearly scaling approach can be used in this case.

This alternative approach is illustrated in simplistic form in Figure 5 using the query sequence from Chromosome 1 that was used before in Figure 4. The illustration shows how the exact position of $g=4$ bp gap (deletion) in the query sequence can be found using simulated annealing type algorithm with $O(n)$ time complexity (similar strategy can be applied to find a position of insertion by placing the gap in the reference instead).

The gap that is inserted initially at an arbitrary position in the hole region of the query is assumed to be able to move freely anywhere in the region. Its position is updated under an influence of attractive force acting on the gap from each mismatch site in the region, thus providing iterative procedure with $O(n)$ time complexity for finding an optimal gap location.

The performance (i.e. rate of convergence) of simulated annealing type optimization is implementation dependent and can be controlled or fine tuned by changing various parameters, e.g. annealing (cooling) schedule (i.e. the rate and the pattern of increase or decrease in the mobility of the gap with respect to the same applied force), type of gap/mismatch interaction potential (e.g. dependence

Table 2. Simulated data. Approximately 25 000 000 bp data of different read lengths and error rates are simulated from the human genome using wgsim program from SAMtools (Li *et al.*, 2009a) using default parameters. These simulated reads are aligned back to the human genome with YOABS, BWA-SW and SSAHA2 (option 454 for 200 bp reads), respectively. The aligned coordinates are then compared with the simulated coordinates to find alignment errors. In each cell in this table, the three numbers are the CPU seconds on a single-core of an AMD Opteron 2356 2.3 GHz CPU, percent alignments with mapping quality ≥20 (Q20), and percent wrong alignments out of Q20 alignments.

Program	Metric	200 bp			500 bp			1000 bp			2000 bp		
		2%	5%	10%	2%	5%	10%	2%	5%	10%	2%	5%	10%
SSAHA2	CPU sec	3398	3928	8309	13 172	29 262	16 980	5177	4806	11 996	8350	5782	9304
	Q20%	95.3	94.8	93.7	96.3	95.9	93.7	97.0	96.8	96.7	97.5	97.5	97.1
	err%	0.00	0.00	0.00	0.00	0.00	0.01	0.00	0.00	0.01	0.00	0.00	0.00
BWA-SW	CPU sec	506	409	304	553	391	363	541	404	306	516	412	300
	Q20%	94.5	89.9	63.9	96.7	95.8	93.0	97.4	97.2	96.5	97.6	97.7	97.5
	err%	0.00	0.01	0.08	0.00	0.00	0.01	0.00	0.00	0.01	0.00	0.00	0.00
YOABS	CPU sec	389	377	432	400	424	382	442	528	440	511	606	499
	Q20%	95.1	91.7	79.9	97.6	96.7	93.6	98.1	97.7	96.9	98.1	98.3	97.8
	err%	0.00	0.01	0.07	0.00	0.00	0.01	0.00	0.00	0.01	0.00	0.00	0.00
YOABS (acc.)	CPU sec	292	347	444	197	287	335	180	321	302	201	329	313
	Q20%	95.1	91.6	79.1	97.6	96.7	93.6	98.1	97.7	96.9	98.1	98.3	97.8
	err%	0.00	0.01	0.07	0.00	0.00	0.01	0.00	0.00	0.01	0.00	0.00	0.00

of the force on the gap/mismatch separation), effective gap mass and so on.

The detailed comparative study of performance and accuracy of the above sketched procedure versus the Smith–Waterman algorithm (as well as study/search for the parameters of simulated annealing that provide the best performance/accuracy trade-off) will be conducted and reported elsewhere. But simple proof of concept used in test code showed promising overall results.

2.4 Acceleration

Although processing the entire query sequence to completely fill the local hit table up (i.e. including all local hits between the query and the reference) seems fast enough to provide computationally competitive implementation, straightforward acceleration has been implemented in the test code.

The acceleration involved introduction of additional hit table and filling both of the tables simultaneously going from the leftmost and the rightmost parts of the query toward each other. The process is terminated as soon as single ungapped or gapped entry in the run-time union of both tables can clearly be identified as a single significant alignment and the rest of the entries is at the level of noise created by random hits. When the query may be aligned at several locations the algorithm will behave exactly like in non-accelerated case and fill the complete local hit table up. For the test cases reported below this resulted in 2–2.5 speedup factor on long low error rate queries without deterioration of accuracy.

3 RESULTS

3.1 Performance evaluation

For the purposes of preliminary testing the YOABS algorithm has been prototyped using $l=m=7$ and $k=8$, thus resulting in ∼2 GB maximum run-time memory usage in all runs reported below.

3.1.1 Mapping quality: The concept of mapping quality based on estimation of the probability of a query sequence being placed at a wrong position has been introduced by Li *et al.* (2008). Both SSAHA2 and BWA-SW reports the mapping quality. To estimate the mapping quality of YOABS alignment an empirical formula, similar to the formula used in BWA-SW has been used: $250\cdot(S1-S2)/S1$, where $S1$ is the score of the best alignment, S2 the score of the second best alignment.

3.1.2 Simulated data: The data was generated using SAMtools package (Li *et al.*, 2009a). Table 2 shows the CPU time, fraction of confidently aligned reads and alignment error rates for BWA-SW (version 0.5.9), SSAHA2 (version 2.5.1) and both non-accelerated and accelerated YOABS given different read lengths (from 200 bp to 2000 bp) and error rates (2%, 5%, 10%). No special optimization or fine tuning of the parameters has been used and in all runs the default command-line options of each aligner have been chosen (except 200 bp SSAHA2 runs, where '−454' option was added).

From Table 2 one can see that non-accelerated test version of YOABS performs comparably to BWA-SW (slightly faster on low error rate reads) and both of them are faster than SSAHA2. On the other hand the accelerated YOABS version is significantly faster than BWA-SW on low error rate reads.

For query lengths >500 bp YOABS shows that performance at both low error rate (2%) and high error rate (10%) ends are better than in the middle (5%) range, especially for accelerated version of the algorithm. In general, this is good property as it allows to characterize the worst performance of YOABS for any given query length. This behavior seems to indicate that local hit table method employed by the algorithm for quickly identifying top hits coupled with acceleration is well suited for low error rate regime and at the same time can be expected to perform reasonably well and do not play the role of show stopper at high error rate limit.

3.1.3 Real data (454): SRR003161 454 dataset (total of 1 376 694 reads) was used for performance evaluation. The results (shown in Table 3) are mostly consistent with the analysis of randomly selected 10% of the reads by Li and Durbin (2010), where ∼1200 plausible different mappings between BWA-SW and SSAHA2 were detected, i.e. either mapped by only one aligner with high score or mapped differently with high score (Q20 or above) by

Table 3. Real data. Total of 1 376 694 reads of 454 dataset SRR003161 were mapped against the human genome using BWA-SW, SSAHA2 and YOABS (both standard and accelerated). The three numbers given for each of the program (first column) are the CPU seconds on a single-core of an AMD Opteron 2356 2.3 GHz CPU, percent alignments with mapping quality ≥20 (Q20) and peak memory (GB) used by the program during the execution.

Program \ Metric	CPU sec	Q20%	Peak memory (GB)
BWA-SW	14 612	88.0	3.7
SSAHA2 [−454]	89 523[a]	90.0[a]	6.7
SSAHA2	538 263[b]	39.9[b]	3.8
YOABS	8868	87.9	2.1
YOABS (acc.)	7080	87.7	2.1

[a]The numbers are estimated, because SSAHA2 crashed in two attempts at or after the record SRR003161.296182. [b]The numbers are estimated as well, the run was terminated after 2 days (at the record SRR003161.406192).

Table 4. Real data. Total of 26 347 reads of PacBio *E.coli* C227-11 filtered reads dataset m110618_035655_42142 were mapped against PacBio *E.coli* C227-11 assembly (37 contigs) using BWA-SW, SSAHA2 and YOABS (accelerated) The two numbers given for each of the program (first column) are the CPU seconds on a single-core of an AMD Opteron 2356 2.3 GHz CPU and percent alignments with mapping quality ≥20 (Q20).

Program \ Metric	CPU sec	Q20%
BWA-SW	911	46.4
SSAHA2	2084	52.0
YOABS (acc.)	570	65.1

at least one aligner). The difference between YOABS and BWA-SW roughly lies below this 0.9% with YOABS accelerated being twice as fast with almost two times less memory used.

3.1.4 Real data (PacBio): Escherichia coli C227-11 filtered reads PacBio dataset m110618_035655_42142 (total of 26 347 reads, ∼82 Mbp) has been processed against PacBio *E.coli* C227-11 assembly (37 contigs) using BWA-SW, SSAHA2 and YOABS. This dataset from a third generation sequencing (PacBio single molecule platform) can be characterized by high indel error rates (typical indel error rate is in the range 10–20%). The mean read length is a little >3 Kbp, with ∼21 Kbp maximum length. Because of on average longer reads YOABS shows good performance on this dataset as well, both in terms of execution time and mapping ratio.

3.2 Overall comparison

Current test implementation of YOABS shows best usage of memory, that can be seen from Table 3.

These preliminary results combined with relatively low algorithmic complexity of YOABS (number of simple look ups tied with simulated annealing versus compressed BWT indexing tied with dynamic programming) clearly indicate compelling nature of the algorithm for use in long sequence realignment projects.

4 CONCLUSION

YOABS is an efficient algorithm (both memory- and performance-wise) for aligning a several hundred or more base pairs query sequence to a long reference genome. It has high sensitivity and specificity (especially given a long query or a query with low error rate). The accuracy of YOABS is comparable with the most accurate long sequence aligners so far (e.g. BWA-SW or SSAHA2). By design YOABS is well suited to detect arbitrary gaps and chimeras, therefore it can be used to facilitate detection of structural variations or reference misassemblies.

In contrast to the majority of long sequence alignment algorithms (e.g. BWA-SW, SSAHA2 or BLAT) YOABS does not use the seed-and-extend paradigm. Instead it records all local hits between all l(prefix)+m(suffix) base pairs index entries for the reference sequence (organized as a forward and a backward tries) and all $l+m$ base pairs subsequences (including l and $2l$ gapped) of the query sequence. The local hits are stored as a table of l-scaled modulo 2^k query subtracted location in the reference versus modulo l location in the query. As a result, the algorithm avoids using the expensive dynamic programming stage (the Smith–Waterman algorithm) altogether replacing it with linearly scaling simulated annealing type of procedure.

The overall complexity of the algorithms is bounded by the complexity of the local hit table building step, that is $O(2^k q)$ (where q is the query length) or $O(256q)$ for $k=8$ used in prototype implementation, hence it does not depend on the size r of the reference. It would be interesting to estimate the value of a constant in the complexity expression but I would expect it to be much lower than $r^{0.628}/256$, taking into account relatively low cost of look-up operations that comprise most of the local hit table construction efforts when comparing with $O(r^{0.628}q)$ complexity expressions for BWA-SW or BWT-SW (Lam *et al.*, 2008; Li and Durbin, 2010), or with $O(rq)$ of the dynamic programming step in general. This low complexity also makes the algorithm compelling for GPU and/or FPGA implementation.

An important difference of the algorithm from various band accelerated modifications of the Smith–Waterman algorithm (that is from approaches maintaining a small fraction of the dynamic programming matrix and, hence, allowing better than $O(rq)$ scaling at the expense of missing some of the possible matches, for example for gaps larger than chosen band size) is that it records all the local hits and therefore will not miss any of the true matches.

Other interesting projects to pursue consist in detailed comparison of performance and accuracy of the simulated annealing stage of the algorithm with the dynamic programming approach for ranges and errors typical for resequencing projects, as well as introducing a query indexing to make it sublinear complexity suitable for *de novo* sequencing.

ACKNOWLEDGEMENT

The author is thankful for stimulating discussions with Valentin Shevchenko. The author is also grateful to the reviewers for a number of important comments and suggestions.

Conflict of Interest: none declared.

REFERENCES

Altschul,S.F. *et al.* (1997) Gapped BLAST and PSI-BLAST: a new generation of protein database search programs. *Nucleic Acids Res.*, **25**, 3389–3402.

Burrows,M. and Wheeler,D.J. (1994) A block-sorting lossless data compression algorithm. *Technical Report 124*. Digital Systems Research Center.

http://www.hpl.hp.com/techreports/Compaq-DEC/SRC-RR-124.pdf.
Kent,W.J. (2002) BLAT–the BLAST-like alignment tool. *Genome Res.*, **12**, 656–664.
Lam,T.W. *et al*. (2008) Compressed indexing and local alignment of DNA. *Bioinformatics*, **24**, 791–797.
Langmead,B. *et al*. (2009) Ultrafast and memory-efficient alignment of short DNA sequences to the human genome. *Genome Biol.*, **10**, R25.
Li,H. and Durbin,R. (2009) Fast and accurate short read alignment with Burrows-Wheeler transform. *Bioinformatics*, **25**, 1754–1760.
Li,H. and Durbin,R. (2010) Fast and accurate long-read alignment with Burrows-Wheeler transform. *Bioinformatics*, **26**, 589–595.
Li,H. *et al*. (2008) Mapping short DNA sequencing reads and calling variants using mapping quality scores. *Genome Res.*, **18**, 1851–1858.
Li,H. *et al*. (2009a) The sequence alignment/map format and SAMtools. *Bioinformatics*, **25**, 2078–2079.
Li,R. *et al*. (2009b) SOAP2: an improved ultrafast tool for short read alignment. *Bioinformatics*, **25**, 1966–1967.
Lunter,G. and Goodson,M. (2011) Stampy: a statistical algorithm for sensitive and fast mapping of Illumina sequence reads. *Genome Res.*, **21**, 936–939.
Ma,B. *et al*. (2002) PatternHunter: faster and more sensitive homology search. *Bioinformatics*, **18**, 440–445.
Morgulis,A. *et al*. (2008) Database indexing for production MegaBLAST searches. *Bioinformatics*, **24**, 1757–1764.
Ning,Z. *et al*. (2001) SSAHA: a fast search method for large DNA databases. *Genome Res.*, **11**, 1725–1729.
Pearson,W.R. and Lipman,D.J. (1988) Improved tools for biological sequence comparison. *Proc. Natl Acad. Sci. USA*, **85**, 2444–2448.
Zhang,Z. *et al*. (2000) A greedy algorithm for aligning DNA sequences. *J. Comput. Biol.*, **7**, 203–214.