Exact Search Algorithms for Biological Sequences

Eric Rivals, Leena Salmela, Jorma Tarhio

CHAPTER 1

EXACT SEARCH ALGORITHMS FOR BIOLOGICAL SEQUENCES

Eric Rivals, Leena Salmela, and Jorma Tarhio

1.1 INTRODUCTION

With the development of sequencing techniques, it has become easy to obtain the sequence, *i.e.* the linear arrangement of residues (nucleotides or amino-acids), of DNA, RNA, or protein molecules. However, determining the function of a molecule remains difficult and is often bound to find a sequence similarity to another molecule whose role in the cell is at least partially known. Then, the biologist can predict that both molecules share the same function and try to check this experimentally. *Functional annotations* are transferred from one sequence to the other provided that their similarity is high enough. This procedure is also applied to molecules subparts, whose sequences are shorter: protein domains, DNA/RNA motifs, etc.

Depending on the sequence lengths and expected level of evolutionary relatedness, the sequence similarity can be found using alignment or pattern matching procedures. A quest in bioinformatics has been to design more sensitive sequence similarity searching methods to push further the limit or *gray*

zone at which evolutionary sequence similarity cannot be departed from random sequence similarity [4, 21]. These methods (*e.g.* profile Hidden Markov Models) have provided, at the expense of computing time, important improvements in functional annotations. However, it has soon become clear that in other frameworks only high level similarity was sought, and speed rather than sensitivity was the major issue. Hence, researchers have designed a continuum of methods that can be classified according to level of allowed dissimilarity:

1. full sensitivity alignment (Smith & Waterman algorithm [58]),
2. fast similarity search programs (*e.g.* BLAST [4]),
3. approximate pattern matching (*e.g.* BOWTIE [36]),
4. near-exact and exact pattern matching (*e.g.* MPSCAN [54]).

For some of everyday sequence manipulation tasks, the user needs exact pattern matching programs (as available in large bioinformatic program suites like EMBOSS): to find back from which chromosome or where in a genome does a given sequence comes from; to find short nucleotidic motifs, like restriction or cleavage sites, in long DNA sequences; to verify whether a *distinguishing* sequence motif really departs negative from positive instances (longer sequences). The latter happens when designing oligonucleotides for gene expression arrays or multiple primers for multiplex polymerase chain reactions [51]. Even for exploring protein sequences, a server has been launched that offers an exact search for short polymers in all sequences of protein data-banks [9]. In such frameworks, the need is single pattern search or multiple pattern search for a few hundreds patterns, which can easily be solved by repetitively applying a single pattern matching program. Algorithmic solutions for these tasks will be described in Section 1.2. However, pattern matching algorithms fail to become popular among biologists for several reasons:

- most of them lack implementations capable of handling biological sequence formats (which then requires to change the format),
- they lack a graphical interface or were not integrated in popular graphical sequence exploration package like the GCG (Genetics Computer Group) package,
- as BLAST [4] was used for similarity searching on a daily basis, it has become the all-purpose tool for most sequence processing tasks, even when more adapted solutions were available [15].

Since 2005, biology experiences the revolution of *high-throughput sequencing* (HTS) due to the renewal of sequencing techniques (new technologies are often termed *Next Generation Sequencing*) [43]. Due to the invention of parallel sequencing of multiple molecules on a single machine, the sequencing output per run has grown by orders of magnitude compared to the traditional

Sanger technique, and is expected to increase further [20]. This change has not only technological consequences. Experiments previously done by hybridisation are now preferentially performed by sequencing [10], since these techniques offer a much deeper sampling and allows to cover the whole genome. Hence, HTS is now exploited to address surprisingly diverse biological questions: genome sequencing or resequencing [43, 5], transcriptomics [59, 50], genomic variation identification or genome breakpoint mapping [14], metagenomics [24], and epigenomics [12]. To grasp how drastic the shift is, consider that one epigenomic census assays published in 2007 produced in one experiment an already amazing 1.5 millions short read sequences of 27 base pairs (bp)[1] each [27], while another published only one year later delivered with the same technology 15 millions 20 bp reads [12].

In all such applications, the first bioinformatic task is to *map* the short reads on a reference genome sequence or on a large collection of DNA sequences. The goal of mapping in transcriptomics, epigenomics and other applications is to point out chromosomic positions either transcribed [59], bound to a protein [27], or whose three dimensional conformation is altered by a protein [12]. Hence, further analysis only consider those reads that mapped to a unique genomic positions. In other frameworks, all mapped reads inclusive those mapped at multiple positions provide important information to detect, *e.g.* new copies of repeats in the sampled genome. The number of (uniquely and/or multi-) mapped reads depends on the read length, on the expected probability for a read to map on the genome, on the level of sequence errors in the reads, as well as on the genetic differences between the cell from which the reads were sequenced and that which provided the reference genome sequence. Two approaches are possible: to map exactly or approximately (up to a limited differences number between the read and the genome sequences) reads on the genome sequence. The choice between the two is not obvious since it has been shown for instance that exact mapping with a shorter read length can yield the same number of uniquely mapped reads than approximate matching up to two mismatches [54], and since all approximate mapping tools are not based on the same algorithm [26, 57, 39, 40, 36]. If approximate mapping is used, another question is then how to distinguish a difference due to a genetic variation or to sequence error in a match?

More practically, whether or not sequence quality information is provided aside the reads themselves, often the complete read sequence cannot be exploited because of low quality positions. Hence, either preprocessing with various parameters is applied to eliminate some positions or multiple mapping with different parameters are tested to optimize the mapping output. In any case, the number of reads to map is so large that mapping efficiency and scalability, both in terms of time and to a less extent of memory, becomes a major issue. In Section 1.4, we will discuss on the comparison of exact

[1] A base pair is the length unit of a DNA/RNA sequence.

versus approximate mapping approaches on these issues. Before that, Section 1.2 presents efficient solutions for the *single pattern matching problem*, while Section 1.3 details fast algorithms for *multiple* or *set pattern matching*.

1.2 SINGLE PATTERN MATCHING ALGORITHMS

We consider exact string matching for locating the occurrences of single nucleotide or amino acid sequence patterns in long biological sequences. We assume that the sequences are in the raw format.

1.2.1 Algorithms for DNA Sequences

Most of the efficient string matching algorithms in the DNA alphabet are modifications of the Boyer–Moore algorithm [11], which applies two heuristics: match and occurrence. Most often only the occurrence heuristic (also called the bad character heuristic) is applied for shifting. The Boyer–Moore–Horspool algorithm [22] (BMH) is the most famous implementation of this simplification. Because the DNA alphabet contains only four symbols, shifts based on one character are short on average. Therefore, it is advantageous to apply q-mers (or q-grams), strings of q characters, for shifting instead of single characters. This technique was already mentioned in the original paper of Boyer and Moore [11, p. 772], and Knuth [34, p. 341] analyzed theoretically its gain. Zhu and Takaoka [65] presented the first algorithm utilizing the idea. Their algorithm uses two characters for indexing a two dimensional array.

Baeza-Yates [6] introduced an extension of the BMH algorithm, where the shift array is indexed with an integer formed from a q-mer with `shift` and `add` instructions. For this kind of approach the practical upper limit with 8 bit characters is two characters.

For the DNA alphabet Kim and Shawe-Taylor [32] introduced a convenient alphabet compression by masking the three lowest bits of ASCII characters. In addition to the `a`, `c`, `g`, and `t` one gets distinguishable codes also for `n` and `u`. Even the important control code \`n`=LF has a distinct value, but \`r`=CR gets the same code as `u`. With this method they were able to use q-mers of up to six characters. Indexing of the shift array is similar to Baeza-Yates' algorithm.

With the DNA alphabet the probability of an arbitrary short q-mer appearing in a long pattern is high. This restricts the average shift length. Kim and Shawe-Taylor [32] introduced a variation for the cases where the q-mer in the text occurs in the pattern. Then two additional characters are checked one by one to achieve a longer shift.

In most cases the q-mer that is taken from the text does not match with the last q-mer of the pattern, and the pattern can be shifted forward. For efficiency one can use a *skip loop* [23], where the pattern is moved forward until the last q-mer of the pattern matches with a q-mer in the text. The easiest

way to implement this idea is to place a copy of the pattern as a stopper after the text and artificially define the shift of the last q-mer of the pattern to be zero. After a skip loop the pattern is compared with the corresponding text positions.

A crucial thing for the efficiency of a q-mer algorithm is how q-mers are computed. Tarhio and Peltola [61] presented a q-mer variation of BMH that applies a skip loop. The algorithm computes an integer called fingerprint from a q-mer. The ASCII codes are mapped to the range of 4: $0 \leq r[x] \leq 3$, where $r[x]$ is the new code of x, such that characters `a`, `c`, `g`, and `t` get different codes and other possible characters get e.g. code 0. In this way the computation is limited to the effective alphabet of four characters. The fingerprint is simply a reversed number of base 4. A separate transformation table h_i is used for each position i of a q-mer and multiplications are incorporated during preprocessing into the tables: $h_i[x] = r[x] \cdot 4^i$. For $q = 4$, the fingerprint of $x_0 \cdots x_3$ is $\sum_{i=0}^{3} r[x_i] \cdot 4^i$, which is then computed as

$$h_0[x_0] + h_1[x_1] + h_2[x_2] + h_3[x_3].$$

Recently Lecroq [37] presented a related algorithm. Its implementation is based on the Wu–Manber algorithm [63] for multiple string matching, but as suggested above, the idea is older [11, 65]. For $q = 4$, the fingerprint of $x_0 \cdots x_3$ is

$$((((((x_0 << 1) + x_1) << 1) + x_2) << 1) + x_3) \bmod 255.$$

SSABS [56] and TVSBS [62] were developed with biological sequences in mind. SSABS is a Boyer–Moore type algorithm. In the search phase the algorithm verifies that the first and last characters of the pattern match with the current alignment before checking the rest of the alignment (a.k.a. guard tests). TVSBS uses a 2-mer for calculating the shift, adopted from the Berry–Ravindran algorithm [8], which is a cross of the Zhu–Takaoka algorithm and Sunday's QS algorithm [60]. Instead of the two-dimensional shift table of Berry–Ravindran, TVSBS uses a hash function to compute an index to a one-dimensional table. According to Kalsi et al. [28], SSABS and TVSBS are not competitive with q-mer algorithms in the DNA alphabet.

We present one fast q-mer algorithm for DNA sequences in detail. It is SBNDM4 [19], which is a tuned version of BNDM (Backward Nondeterministic DAWG Matching) by Navarro and Raffinot [45]. BNDM is a kind of cross of the Backward DAWG Matching algorithm (BDM) [16] and Shift-Or [7] algorithms. The idea of BNDM is similar as in BDM, while instead of building a deterministic automaton, a nondeterministic automaton is simulated even without constructing it. The resulting code applies bit-parallelism and it is efficient and compact. We present a C code for SBNDM4 as Alg. 1.1.

Algorithm 1.1

```
01  for  (i  =  0;  i  <  256;  i++)  B[i]  =  0;
02  for  (i  =  0;  i  <  m;  i++)  B[P[m-i]]  |=  (1<<i);
03  for  (i  =  0;  i  <  m;  i++)  T[n+i]  =  P[i];
04  while  (1)  {
05      while  (!(d=(B[T[j]]<<3)&(B[T[j-1]]<<2)&(B[T[j-2]]<<1)&B[T[j-3]]))
06          j  +=  m-3;
07      pos  =  j;
08      while  (d=(d<<1)&B[T[j-4]])  j--;
09      j  +=  m-4;
10      if  (j  ==  pos)  {
11          if  (j  >=  n)  return  (nmatch);
12          nmatch++;
13          j++;
14      }
15  }
```

There `P[0],...,P[m-1]` is the pattern and `T[0],...,T[n-1]` the text. The code contains a skip loop so that a copy of the pattern is placed to `T[n],...,T[n+m-1]`. This version prints only the number of matches. The matches can be reported by changing line 12.

SBNDM4 is very fast in practice. One x86 processors one can still boost its performance by using 16-bit reading [19]. In searching DNA patterns of 20 characters, SBNDM4 with 16-bit reading is more than eight times faster than the classical Boyer–Moore algorithm [19] (see also comparisons [28, 37, 61]).

SBNDM4 works for DNA patterns of up to 32 or 64 characters depending on the word size of the processor. In practice longer exact patterns are seldom interesting, but e.g. Lecroq's algorithm is good for them.

SBNDM4, like other variations of BNDM, works also for more general string matching where positions in the pattern or in the text represent character classes [46] instead of single characters. So e.g. the standard IUB/IUPAC nucleic acid codes can be used with SBNDM4.

There are also algorithms [52, 33] for packed DNA. We decided to leave them outside this presentation.

1.2.2 Algorithms for Amino Acids

In general, there is not much difference in searching amino acid and natural language patterns. So any good search algorithm for natural language is also applicable to amino acids. In searching short patterns [19], the 2-mer variation of SBNMD4 is among the best. SBNDM2 is got from SBNDM4 as follows: replace line 5 by

```
while (!(d=(B[T[j]]<<1)&B[T[j-1]]))
```

and $m-3$ by $m-1$, $j-4$ by $j-2$, and $m-4$ by $m-2$ on lines 6–9.

As in the case of SBNDM4 one can still boost the performance of SBNDM2 by using 16-bit reading [19]. In searching patterns of 5 characters, SBNDM2

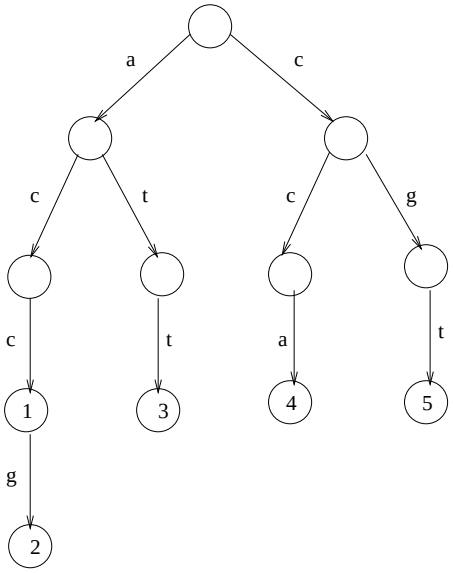


Figure 1.1: An example trie storing the strings {*acc, accgt, attg, ccat, cgtt*}

with 16-bit reading is more than two times faster than the classical Boyer–Moore algorithm.

1.3 ALGORITHMS FOR MULTIPLE PATTERNS

In this section, we consider exact searching of multiple patterns. More precisely, we are given a text and r patterns and we need to find all occurrences of all the patterns in the text.

1.3.1 Trie-Based Algorithms

Many algorithms for exact searching of multiple patterns are based on a data structure called *trie* for storing the patterns. A trie is a tree where each edge is labeled with a character. Each node of the trie is associated with a string that is formed by concatenating all the labels of the edges on the path from the root to the node. Given a node in the trie, all edges to the children of this node have a different label. Figure 1.1 shows the trie storing the strings {*acc, accgt, attg, ccat, cgtt*}.

1.3.1.1 Aho-Corasick The Aho-Corasick algorithm [2] builds as preprocessing an automaton that recognizes the occurrences of all patterns. The preprocessing starts by building the trie of the pattern set. We add an edge from the root to the root for all those characters that do not yet have an outgoing edge

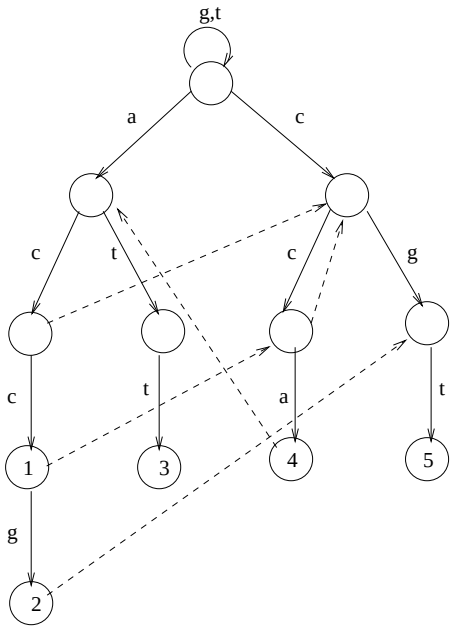


Figure 1.2: An example of the Aho-Corasick automaton for the patterns {*acc, accg, att, cca, cgt*}. The dashed lines show the failure links. Failure links to the starting state of the automaton have been omitted.

from the root. The trie is then augmented with failure links as follows. The failure link of a node N in the trie points to a node that is associated with the longest possible suffix of the string associated with the node N excluding the node N itself. Additionally we associate an output function with each node whose associated string is one of the patterns. The output function outputs the identifier of the pattern. Figure 1.2 shows an example of an Aho-Corasick automaton for the patterns {*acc, accg, att, cca, cgt*}.

The automaton is used for searching the text as follows. We start at the root node of the trie. We read the text character by character and for each character we perform the following actions. While there is no child node with an edge labeled with the read character, we follow the failure link. Then we descend to the child with an edge labeled with the read character. Finally we output the identifiers returned by the output function for the child node.

1.3.1.2 Set Backward Oracle Matching The Set Backward Oracle Matching (SBOM) algorithm [3] builds an automaton that recognizes at least all factors of the reversed patterns. The automaton is built as follows. First we build a trie of the reversed patterns. Then we traverse the trie in breath-first order and add some more edges between the nodes turning the trie into a directed acyclic graph (DAG) that recognizes all factors of the reversed patterns. To

sbom_preprocess $(P_1, \ldots, P_r)$

```
1.  build_trie(P_1^r, ..., P_r^r)
2.  set supply link of root to NULL
3.  for each node N in the trie in breath first order
4.      down = supply link of parent
5.      c = the edge label from parent to N
6.      while (down ≠ NULL and down does not have a child with edge label c)
7.          add an edge from down to N with label c
8.          down = supply link of down
9.      if (down ≠ NULL)
10.         set supply link of N to the child of down with edge label c
11.     else
12.         set supply link of N to root
```

Figure 1.3: Preprocessing of SBOM

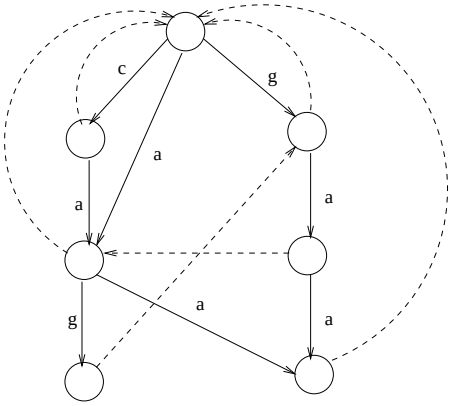


Figure 1.4: An example of the SBOM automaton. The dashed lines show the supply links.

assist us in adding these new edges we associate a supply link with each node. For each node we then perform the pseudo-code on lines 4 to 12 shown in Figure 1.3. Figure 1.4 shows an example of the SBOM automaton built for patterns $\{aag,\ gac\}$ As we can see the automaton recognizes also some other strings, like *caa*, than the factors of the patterns.

This automaton is then used for searching the occurrences of the patterns as follows. Initially we set the endpoint to the length of the shortest pattern. We then read the characters of the text backward starting at the endpoint character. For each character we make the corresponding state transition in

the automaton. Whenever we encounter a node associated with one of the patterns, we verify the read region character by character against the pattern. If we encounter a character that does not have a transition on that character, we can shift the endpoint forward and start the backward scan again at the new endpoint. The length of the shift is $m - j$, where m is the length of the shortest pattern and j is the number of characters that we have read, or the length of the shortest pattern, whichever is shorter.

1.3.2 Filtering Algorithms

Filtration aims at eliminating most positions that cannot match any pattern with an easy criterion. Then, verification checks whether the remaining positions truly match a pattern. Thus filtering algorithms operate in three phases. The patterns are first preprocessed, in the second phase we search the text with a filtering method, and the candidate matches produced by the filtering are verified in the third phase.

Here we describe several algorithms that use a generalized pattern of character classes for filtration [55]. Let us explain the filtration scheme with an example. Assume a set of 3 patterns of length $m = 8$: $\{P_1, P_2, P_3\} = \{accttggc, gtcttggc, accttcca\}$, and set q to 5. The overlapping 5-mers (or 5-grams) of each pattern are given in Figure 1.5. For a text window W of length 8 to match P_1, the substring of length q starting at position i in W must match the i^{th} q-mer of P_1 for all possible i, and conversely. Now, we want to filter out windows that do not match any pattern. If the substring starting at position i in W does not match the i^{th} q-mer of neither P_1, P_2, nor P_3, then we are sure W cannot match any of the patterns. Thus, our filtration criterion to surely eliminate any non-matching window W is to find if there exists a position i such that the previous condition is true.

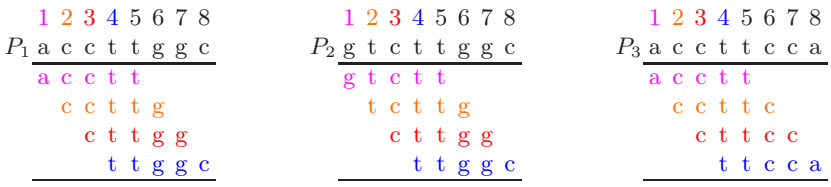
Given a set of patterns, the filtering algorithms build a single q-mer generalized pattern (Fig. 1.5c). A generalised pattern allows several symbols to match at a position (like a position $[DENQ]$ in a PROSITE pattern, which matches the symbols D, E, N, and Q). However, here each q-mer is processed as a single symbol. Then, a string matching algorithm that can handle classes of characters is used for searching for occurrences of the generalized pattern in the text.

Various different algorithms can be used for implementing the filtering phase. Below we describe in more detail algorithms where filtering is based on the shift-or, BNDM, and Boyer-Moore-Horspool algorithms. A filtering algorithm always requires an exact algorithm to verify the candidate matches. In principle, any of the presented exact algorithms could be used for this purpose.

1.3.2.1 Multi-Pattern Shift-Or with q-Grams The shift-or algorithm is easily extended to handle classes of characters in the pattern [1, 7], and thus developing a filtering algorithm for multiple pattern matching is straightforward. The

$$\{P_1, P_2, P_3\} = \{accttggc, gtcttggc, accttcca\}$$

(a)



(b)

$$[acctt, gtctt][ccttg, tcttg, ccttc][cttgg, cttcc][ttggc, ttcca]$$

(c)

Figure 1.5: (a) A set of 3 patterns of length $m = 8$. (b) The overlapping 5-mers starting at position 1 to 4 (resp. in magenta, orange, red, blue) of each pattern. (c) The generalised 5-mer pattern for the set of tags.

preprocessing phase now initializes the bit vectors for each q-mer as follows. The i:th bit is set to 0 if the given q-mer is included in the character class in the i:th position. Otherwise the bit is set to 1. The filtering phase proceeds then exactly like the matching phase of the shift-or algorithm. Given this scheme, it is clear that all actual occurrences of the patterns in the text are candidates. However, there are also false positives as the generalized pattern matches also other strings than the original patterns.

1.3.2.2 Multi-Pattern BNDM with q-Grams The second filtering algorithm is based on the BNDM algorithm by Navarro and Raffinot [45]. This algorithm has been extended to classes of characters in the same way as the shift-or algorithm. We call the resulting multiple pattern filtering algorithm BG (short for BNDM with q-Grams). The bit vectors of the BNDM algorithm are initialized in the preprocessing phase so that the i:th bit is 1 if the corresponding q-mer is included in the character class of the reversed generalized pattern in position i. In the filtering phase, the matching is then done with these bit vectors. As with SOG, all match candidates reported by this algorithm must be verified.

1.3.2.3 Multi-Pattern Horspool with q-Grams The last of our algorithms uses a Boyer-Moore-Horspool [22] type method for matching the generalized pattern against the text. Strictly speaking, this algorithm does not handle character classes properly. It will return all those positions where the generalized pattern matches and also some others. This algorithm is called HG (short for Horspool with q-Grams).

5-mer tables:

1.	2.	3.	4.
accct	accct	accct	accct
	ccctt	ccctt	ccctt
		cctta	cctta
			cttaa

(a)

hg_matcher $(T = t_1 \dots t_n,\, n)$

```
1.  i = 1
2.  while(i ≤ n − m + 1)
3.      j = m − q + 1
4.      while (1)
5.          if (not qMerTable[j][t_{i+j−1...i+j+q−2}])
6.              i = i + j
7.              break
8.          else if (j = 1)
9.              verify_match (i)
10.             i = i + 1
11.             break
12.         else
13.             j = j − 1
```

(b)

Figure 1.6: The HG algorithm: (a) the data structures for the pattern "acccttaa" and (b) the pseudo code for the search phase.

The preprocessing phase of HG constructs a bit table for each of the $m-q+1$ positions where a q-mer starts in the pattern. The first table keeps track of q-mers contained in the character class of the first position of the generalized pattern, the second table keeps track of q-mers contained in the character classes of the first and the second position in the generalized pattern, and so on. Finally, the $m-q+1$:st table keeps track of characters contained in any of the character classes of the generalized pattern. Figure 1.6a shows the four tables corresponding to the pattern 'acccttaa' when using 5-mers.

These tables can then be used in the filtering phase as follows. First, the $m-q+1^{\text{st}}$ q-mer is compared with the $m-q+1^{\text{st}}$ table. If the q-mer does not appear in this table, the q-mer cannot be contained in the character classes of positions $1 \dots m-q+1$ in the generalized pattern, and a shift of $m-q+1$ characters can be made. If the character is found in this table, the $m-q$:th character is compared to the $m-q$:th table. A shift of $m-q$ characters can

be made if the character does not appear in this table and therefore not in any character class in the generalized pattern in positions $1, \ldots, m-q$. This process is continued until the algorithm has advanced to the first table and found a match candidate there. The pseudo code is shown in Figure 1.6b. Given this procedure, it is clear that all positions matching the generalized pattern are found. However, also other strings will be reported as matches.

1.3.3 Other Algorithms

Other algorithms for searching multiple patterns include the Commentz-Walter algorithm [17] with its variations, the Wu-Manber algorithm [63], and algorithms derived from the Rabin-Karp algorithm [29] for a single pattern.

1.4 APPLICATION OF EXACT SET PATTERN MATCHING FOR READ MAPPING AND COMPARISON WITH MAPPING TOOLS

Here, we concentrate on the question of set pattern matching and on its main current application: read mapping on genomic sequences. In most frameworks, millions of reads that originate from a genome have been sequenced using HTS. A read serves as a signature for a molecule or a chromosomic position. The goal of mapping is to find back for each different read the chromosomic position of origin in the reference genome. As a read may be sequenced several time according to its number of occurrence in the biological sample, the number of different reads may be much lower than the number of read sequences. For example, in a transcriptomic assay were 2 million reads were sequenced, the read **set** contains $\simeq$ 440.000 elements [50]. As a read sequence can differ from the original chromosomic sequence because of polymorphisms or sequence errors, read mapping is often performed using approximate pattern matching, which allows for a few mismatches and or indels. For approximate mapping, either near-exact sequence similarity search programs ((BLAT [30], MegaBlast [64], or SSAHA [48]) or mapping tools (Eland, tagger [25], rmap [57], seqmap [26], SOAP [39], MAQ [38], Bowtie [36], and ZOOM [40]) are used. An alternative option when dealing with short reads is to resort to exact set pattern matching, for which mpscan offers an efficient solution [54, 50].

Due the number of reads to match, repeated application of a single pattern matching algorithm for each read would require an unaffordable computing time. Hence, practically efficient solutions involve

1. either indexing the reads in main memory and scanning the genome only once (or a few times) for all reads (the solution chosen in MegaBlast [64], seqmap [26], or mpscan [54])

2. or first preprocessing the genome to build an index and then loading the index in memory before searching each read one after the other (the

approach followed in SSAHA [48], BLAT [30], and in mapping tools like ELAND, TAGGER [25], RMAP [57], SOAP [39], MAQ [38], BOWTIE [36]).

1.4.1 mpscan: an efficient exact set pattern matching tool for DNA/RNA sequences

The program MPSCAN [54, 50] is an implementation of the exact Multi-Pattern BNDM with q-Grams algorithm (cf. Section 1.3.2.2, also called BG algorithm in [55]). It is specialised for searching large sets of relatively short DNA/RNA patterns in large DNA/RNA sequence files, and its interface is adequate for the purpose of mapping reads: it handles file formats commonly used in biology, can search for the reverse complementary of the pattern, etc.

Its correctness, which ensures it to yield for each read all text positions at which that read matches the text, derives from that of the Multi-Pattern BNDM with q-Grams algorithm [55]. The filtration efficiency depends on the parameter q. We have shown recently that the average time complexity of MPSCAN for searching r patterns of size l in a text of length n over an alphabet of size c is

$$\mathcal{O}(n \log_c(rl)/l) \text{ provided that } q = \Theta(\log_c(rl)).$$

As it was proved that the minimum time required is $\Omega(n \log_c(rl)/l)$ [44], this makes MPSCAN asymptotically optimal in average.

For example on an Intel Xeon CPU 5140 processor at 2.33 GHz with 8 GB main memory, when searching 4 million 27 bp reads on the 247 Mbp of human chromosome 1, MPSCAN sets the parameter q to 13, uses 229 megabytes memory, and takes 78 seconds.

1.4.2 Other solutions for mapping reads

With HTS becoming more popular and the increase of their sequencing capacity, the question of mapping reads on a genome sequence is a crucial issue, as well as a bottleneck.

At the HTS advent, an available solution was to use ultrafast similarity search BLAST-like programs, which were not designed for this purpose, but for locally aligning sequences that differ little (for instance, only because of sequencing errors). There were typically intended to align Expressed Sequence Tags on the human genome. These programs are not adapted to short reads (below 60 bp), and because of internal limitation cannot handle millions of queries. Hence, both their sensitivity and scalability are insufficient for mapping application with short reads [54]. However, some user still resort to these tools for they allow, unlike mapping tools, an unrestricted number of differences between the read and the genome [31]. All these tools implement a filtration strategy that requires a substring of the query sequence to match the genome either exactly [48, 64] or with at most one mismatch [30].

Since the commercialisation of HTS, numbers of commercial or free mapping tools have been developped or published (cf. list above); for instance the ELAND software is provided with the Illumina®Solexa sequencer. As mentioned the goal of mapping differs with the application but it is often to find the best match for a read: the match with the least differences, and if possible unique. All mapping programs perform successive approximate pattern matching up to a limited number of differences. Some tools can find matches with up to 4 mismatches and/or indels, but generally a guarantee to find all matches (as required in the definition of approximate matching) is given only up to one or two mismatches. This limitation makes sense to speed up the search and derives from the applied filtration scheme. All tools (except ZOOM) use variant of the so-called PEX filter [46], which consists in splitting the read in $k+1$ adjacent pieces, knowing that at least one piece will match exactly when a maximum of k errors are allowed. Many mapping programs makes it efficient by using 2-bit encoded sequences and/or an index of the genome (*e.g.*, MAQ, ELAND, BOWTIE, RMAP, SOAP).

The program ZOOM exploits *spaced seeds*: it requires that a subsequence of a defined form, instead of a substring, matches between the read and the genome [13, 41]. The subsequence's pattern of required matching positions and wild-cards is designed on purpose depending on the expected match length and maximal number of differences [35]. The advantage of spaced seeds is their capacity to handle mismatches and insertion/deletions (*indels*), and their increase sensitivity compared to substring based filtration [13, 41]. Their main drawback is the difficulty of seed design: ZOOM uses a conjunction of several seeds. Hence, sets of spaced seeds are specifically designed for a certain read/match length and a maximum number of allowed differences, and different sets corresponding to different parameter combinations are hard coded in ZOOM. All known formulations of the seed design problem are at least NP-hard, even for a single seed [35, 42, 47].

1.4.3 Comparison of mapping solutions

As already mentioned, many groups have developed and/or published their own mapping tools, and all tools, except MPSCAN, implement a solution based on approximate pattern matching. However to date, one lacks a comparative evaluation of the sensibility of all these tools in various application frameworks. The intended application makes a difference since for *e.g.*, identifying genomic variations, multiple matching locations of a read provides useful information, while in transcriptomics one usually discards multi-mapped reads. Probing the sensitivity and evaluating the sensitivity vs speed or memory balance is a difficult task knowing that the programs differ in their notion of approximation (*e.g.*, with or without indels).

Here, we discuss the conclusions of a comparison on the less difficult task of exact set pattern matching. We exclude the program ELAND for it is not

free for academics, as well as MAQ, which does not accept parameters for searching only exact read matches.

1.4.3.1 Speed, memory footprint, and scalability We compared RMAP, SEQMAP, SOAP (v1 & v2), ZOOM, BOWTIE and MPSCAN for searching increasing read sets on the longest human chromosome (chromosome 1, 247 Mbp). The public input data sets contains 6.5 million of 27 bp reads and we took subsets every million reads (available on the GEO database under accession number GSM325934). At the date of this comparison, this set belongs to the largest ones in terms of number of different reads, and there is no available data set of similar size with much larger reads (say > 36).

Figure 1.7 reports the running times in seconds on a logarithmic scale for searching the subsets of $1, 2, \ldots$ up to 6 and 6.5 million reads. Of course the times do not include the index construction time for those programs that use an index, which in the case of, *e.g.* BOWTIE, lasts hours for the complete human genome.

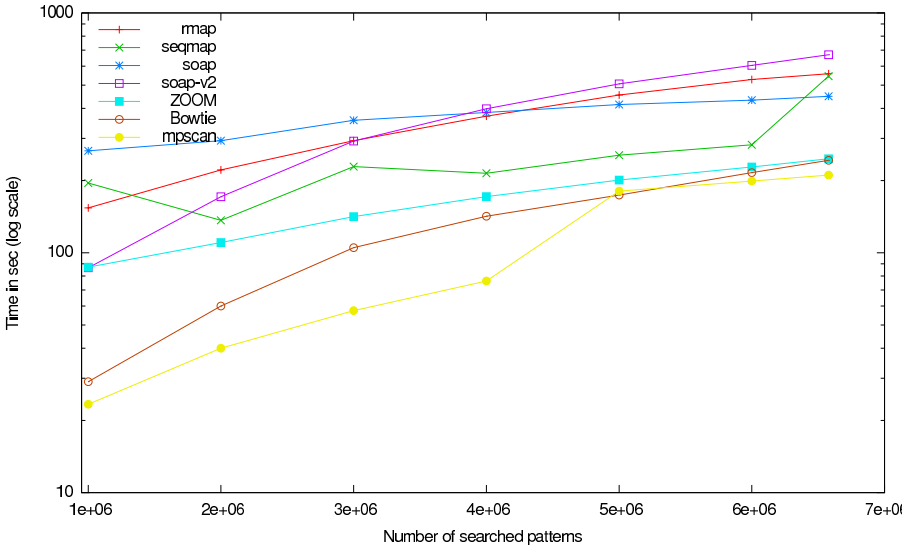


Figure 1.7: Comparison of mapping tools: Search times of RMAP, SEQMAP, SOAP (v1 & v2), ZOOM, BOWTIE and MPSCAN in seconds (log scale) for increasing subsets of 27 bp reads. All tools behave similarly and offer acceptable scalability. MPSCAN remains the most efficient of all, and can be 10 times faster than tools like SEQMAP or RMAP. Times do not include the index construction time.

First, all tools can handle very large read sets and their running times remain impressive even if they degrade somehow with increasing read sets. Second, the comparison of ZOOM or MPSCAN compared to genome indexing tools like BOWTIE or SOAP shows that high performances are not bound to

a genome index, at least for exact pattern matching. Knowing that ZOOM algorithm also handles approximate matches with up to two mismatches or indels, it seems that it offers a very satisfying solution compared to BOWTIE, which is limited to mismatches and offers less guarantee. For exact pattern matching, the performance differences can be quite large (times 10 between MPSCAN and SOAP-V2 for two million reads), and MPSCAN offers the fastest solution overall, even if it exploits only a 32-bit architecture. However, MPSCAN time increases more when going from 4 to 5 million reads, suggesting that for equal read length, a coarse grain parallelization would improve its performances.

To illustrate the low memory footprint of mapping tools that do not load a genome index in RAM, we give the amount of RAM required by ZOOM, SEQMAP, and MPSCAN for searching the complete human genome with one million 27 bp tags. ZOOM requires 17 minutes and 0.9 Gigabytes, RMAP takes 30 min and 0.6 Gb, SEQMAP performs the task in 14 min with 9 Gb, while MPSCAN needs < 5 min using 0.3 Gb. In contrast, BOWTIE human genome index, which is implemented as a Burrows-Wheeler Transform, takes at least 1.4 Gb [36].

1.4.3.2 Exact pattern matching for read mapping The read length influences the probability of a read to map on the genome, and also its probability to map once. The shorter the read the higher the probability of mapping, but the lower that of mapping once. In many applications, reads mapping at unique genomic positions are preferred. A rationale for the currently developped extension of read length is the increase probability to map a unique genomic location. On the human genome, a length of 19 bp already brings the risk of mapping at random below 1% and we have shown recently that it already maximise the number of uniquely mapped reads on four real data sets [50]. Studying the sequence error position in the reads, we could show that the error probability at one sequence position increases with the position in the read for Illumina®/Solexa data. Hence, an alternative to approximate mapping is to perform exact matching using only a prefix of each read (of an adequate length).

To evaluate this we compared the result of approximate matching with full length reads with that of MPSCAN on read prefixes. ELAND searches the best read match up to two mismatches, while we ran MPSCAN to search for exact matches of read prefixes. The full length read are 34 bp. If one maps with MPSCAN the full length reads, 86% remain unmapped and 11% are uniquely mapped. With at most two mismatches, ELAND finds 14% of additional uniquely mapped reads with one or two mismatches, while mapping the 20 bp prefix of each read with MPSCAN allows to map 25% of all reads at unique positions (14% more sites than with full length reads). Hence, both approaches yield similar output, but exact matches represent easier and more secure information than approximate matches. For the current rates of sequencing errors and read lengths, exact matching is a suitable solution

for read mapping. Moreover, it allows us to estimate computationally the sequence error rate without performing control experiments (cf. [50] for a more in-depth presentation), which would be more difficult using approximate matching.

1.4.4 Conclusions

About pattern matching, q-gram based algorithms and especially MPSCAN represent the most efficient theoretical and practical solutions to exact set pattern matching for huge pattern sets (above a million patterns). Compared to known solutions surveyed seven years ago in [46], which were reported to handle several hundred thousands patterns, MPSCAN provides more than an order of magnitude improvement: it allows to process at astonishing speed pattern sets of several millions reads. The second take home message is that its filtration scheme can compete with approaches that use a text index.

Since 2005, the capacity of HTS is continuously evolving: biotechnological research and development aim at reducing the quantity of biological extract, augmenting the sequencing capacity and quality, raising the read length, and even enlarging the application fields. Despite the efforts for designing scalable and efficient mapping programs, it will remain a computational issue to let mapping solutions fit the requirements of new HTS versions. This is a complex question since read length above 20 are not necessary to point out a unique position in a genome as large as that of human [50].

An interesting conclusion is that different filtration schemes achieve impressive efficiency and scalability, but may be insufficient for tomorrow's needs. The abundant pattern matching literature may still contain other possible algorithms whose applications in this setup has not been yet evaluated. With the spread of multi-core computers, parallelization represents another future line of research.

Finally, we left aside the problem of mapping pairs of reads. In this framework, two reads are sequenced for each targeted molecule: each at a different extremity. The reads come in pairs and the goal of mapping is to find one matching position for each read such that the two positions are on the same chromosome and in an upper bounded vicinity. In other applications, the pair relations are unknown and it is then required to find across the two sets of beginning and ending reads, which ones constitute a pair for they map on the same chromosome not too far from another [53]. Some mapping tools like MAQ or ZOOM can solve read pair mapping efficiently, while a precursor of MPSCAN has been developped and applied in the second framework [53].

REFERENCES

1. K. Abrahamson. Generalized string matching. *SIAM Journal on Computing*, 16(6):1039–1051, 1987.

2. A.V. Aho and M.J. Corasick. Efficient string matching: An aid to bibliographic search. *Communications of the ACM*, 18(6):333–340, 1975.

3. C. Allauzen and M. Raffinot. Factor oracle of a set of words. Technical Report 99-11, Institut Gaspard-Monge, Universit de Marne-la-Valle, 1999. (in French).

4. S. F. Altschul, T. L. Madden, A. A. Schaffer, J. Zhang, Z. Zhang, W. Miller, and D. J. Lipman. Gapped blast and psi-blast: a new generation of protein database search programs. *Nucleic Acids Res*, 25:3389–3402, 1997.

5. Jean-Marc Aury, Corinne Cruaud, Valerie Barbe, Odile Rogier, Sophie Mangenot, Gaelle Samson, Julie Poulain, Veronique Anthouard, Claude Scarpelli, Francois Artiguenave, and Patrick Wincker. High quality draft sequences for prokaryotic genomes using a mix of new sequencing technologies. *BMC Genomics*, 9(1):603, 2008.

6. R. Baeza-Yates. Improved string searching. *Software: Practice and Experience*, **19**(3):257–271, 1989.

7. R. A. Baeza-Yates and G. H. Gonnet. A new approach to text searching. *Commun. ACM*, 35(10):74–82, 1992.

8. T. Berry and S. Ravindran. A fast string matching algorithm and experimental results. *Proc. of the Prague Stringology Club Workshop '99*, Czech Technical University, Prague, Czech Republic, Collaborative Report DC-99-05, pp. 16–28, 1999.

9. A.J. Bleasby, D. Akrigg, and T.K. Attwood. OWL - A non-redundant, composite protein sequence database. *Nucleic Acids Research*, 22(17):3574–3577, 1994.

10. Nathan Blow. Transcriptomics: The digital generation. *Nature*, 458:239–242, 2009.

11. R.S. Boyer and J S. Moore. A fast string searching algorithm. *Communications of the ACM*, **20**(10):762–772, 1977.

12. Alan P. Boyle, Sean Davis, Hennady P. Shulha, Paul Meltzer, Elliott H. Margulies, Zhiping Weng, Terrence S. Furey, and Gregory E. Crawford. High-Resolution Mapping and Characterization of Open Chromatin across the Genome. *Cell*, 132:311–322, Jan 2008.

13. S. Burkhardt and J. Kärkkäinen. Better filtering with gapped q-grams. *Fundamenta Informaticae*, 56(1–2):51–70, 2003.

14. Wei Chen, Vera Kalscheuer, Andreas Tzschach, Corinna Menzel, Reinhard Ullmann, Marcel Holger Schulz, Fikret Erdogan, Na Li, Zofia Kijas, Ger Arkesteijn, Isidora Lopez Pajares, Margret Goetz-Sothmann, Uwe Heinrich, Imma Rost, Andreas Dufke, Ute Grasshoff, Birgitta Glaeser, Martin Vingron, and H. Hilger Ropers. Mapping translocation breakpoints by next-generation sequencing. *Genome Research*, 18(7):1143–1149, 2008.

15. J.M. Claverie and C. Notredame. *Bioinformatics for dummies*. Wiley Publishing, Inc., 2003.

16. M. Crochemore and W. Rytter. *Text algorithms*. Oxford University Press, 1994.

17. B. Commentz-Walter. A string matching algorithm fast on the average. In *Proceedings of the 6th Colloquium on Automata, Languages and Programming (ICALP'79)*, volume 71 of *LNCS*, pages 118–132. Springer-Verlag, 1979.

18. K. Fredriksson and Sz. Grabowski. Practical and optimal string matching. In *Proc SPIRE'05, Lecture Notes in Computer Science* **3772**:376–387, 2005.

19. B. Durian, J. Holub, H. Peltola, and J. Tarhio. Tuning BNDM with q-grams. In: Proc. ALENEX '09, Tenth Workshop on Algorithm Engineering and Experiments. SIAM 2009, 29–37.

20. Editor. Prepare for the deluge. *Nat Biotech*, 26:1099, 2008.

21. Dan Gusfield. *Algorithms on Strings, Trees and Sequences*. Cambridge University Press, 1997.

22. R.N. Horspool. Practical fast searching in strings. *Software: Practice and Experience*, **10**(6):501–506, 1980.

23. A. Hume and D. Sunday. Fast string searching. *Software: Practice and Experience*, **21**(11):1221–1248, 1991.

24. Daniel Huson, Daniel Richter, Suparna Mitra, Alexander Auch, and Stephan Schuster. Methods for comparative metagenomics. *BMC Bioinformatics*, 10(Suppl 1):S12, 2009.

25. C Iseli, G Ambrosini, P Bucher, and CV Jongeneel. Indexing Strategies for Rapid Searches of Short Words in Genome Sequences. *PLoS ONE*, 2(6):e579, 2007.

26. Hui Jiang and Wing Hung Wong. Seqmap: mapping massive amount of oligonucleotides to the genome. *Bioinformatics*, 24(20):2395–6, 2008.

27. David S. Johnson, Ali Mortazavi, Richard M. Myers, and Barbara Wold. Genome-Wide Mapping of in Vivo Protein-DNA Interactions. *Science*, 316(5830):1497–1502, 2007.

28. P. Kalsi, H. Peltola, J. Tarhio. Comparison of exact string matching algorithms for biological sequences. In: Proc. BIRD '08, 2nd International Conference on Bioinformatics Research and Development (ed. M. Elloumi et al.). Communications in Computer and Information Science 13, Springer 2008, 417–426.

29. R.M. Karp and M.O. Rabin. Efficient randomized pattern-matching algorithms. *IBM Journal of Research and Development*, 31(2):249–260, 1987.

30. James W. Kent. BLAT—The BLAST-Like Alignment Tool. *Genome Res.*, 12(4):656–664, 2002.

31. Peter V Kharchenko, Michael Y Tolstorukov, and Peter J Park. Design and analysis of ChIP-seq experiments for DNA-binding proteins. *Nat Biotech*, 26(12):1351–9, Dec 2008.

32. J.Y. Kim and J. Shawe-Taylor. Fast string matching using an n-gram algorithm. *Software: Practice and Experience*, **24**(1):79–88, 1994.

33. J.W. Kim, E. Kim, and K. Park. Fast matching method for DNA sequences. In *Proc ESCAPE 2007, Lecture Notes in Computer Science* **4614**:271–281, 2007.

34. D.E. Knuth, J.H. Morris, and V.R. Pratt. Fast pattern matching in strings. *SIAM Journal on Computing*, **6**(1): 323–350, 1977.

35. G. Kucherov, L. Noé, and M. Roytberg. Multiseed Lossless Filtration. *IEEE/ACM Transactions on Computational Biology and Bioinformatics*, 2(1):51–61, Jan-Mar 2005.

36. Ben Langmead, Cole Trapnell, Mihai Pop, and Steven Salzberg. Ultrafast and memory-efficient alignment of short dna sequences to the human genome. *Genome Biology*, 10(3):R25, 2009.

37. T. Lecroq. Fast exact string matching algorithms. *Information Processing Letters*, **102**(6): 229–235, 2007.

38. Heng Li, Jue Ruan, and Richard Durbin. Mapping short DNA sequencing reads and calling variants using mapping quality scores. *Genome Res.*, 18:1851–1858, 2008. in press.

39. Ruiqiang Li, Yingrui Li, Karsten Kristiansen, and Jun Wang. SOAP: short oligonucleotide alignment program. *Bioinformatics*, 24(5):713–714, 2008.

40. Hao Lin, Zefeng Zhang, Michael Q. Zhang, Bin Ma, and Ming Li. ZOOM! Zillions of oligos mapped. *Bioinformatics*, 24(21):2431–2437, 2008.

41. B. Ma, J. Tromp, and M. Li. PatternHunter: faster and more sensitive homology search. *Bioinformatics*, 18(3):440–445, 2002.

42. Bin Ma and Ming Li. On the complexity of the spaced seeds. *J. of Computer and System Sciences*, 73(7):1024–1034, Nov 2007.

43. M Margulies, M Egholm, WE Altman, and S et al. Attiya. Genome sequencing in microfabricated high-density picolitre reactors. *Nature*, 437:376–380, 2005.

44. Gonzalo Navarro and Kimmo Fredriksson. Average complexity of exact and approximate multiple string matching. *Theoretical Computer Science*, 321(2-3):283–290, 2004.

45. G. Navarro and M. Raffinot. Fast and flexible string matching by combining bit-parallelism and suffix automata. *ACM Journal of Experimental Algorithms*, **5**(4):1–36, 2000.

46. G. Navarro and M. Raffinot. *Flexible Pattern Matching in Strings – Practical on-line search algorithms for texts and biological sequences*. Cambridge University Press, 2002.

47. François Nicolas and Eric Rivals. Hardness of optimal spaced seed design. *J. of Computer and System Sciences*, 74:831–849, 2008.

48. Zemin Ning, Anthony J. Cox, and James C. Mulikin. SSAHA: A Fast Search Method for large DNA Databases. *Genome Res.*, 11:1725–1729, 2001.

49. H. Peltola and J. Tarhio. Alternative algorithms for bit-parallel string matching. In *Proc SPIRE'03, Lecture Notes in Computer Science* **2857**:80–93, 2003.

50. N. Philippe, A. Boureux, J. Tarhio, L. Bréhélin, T. Commes, and E. Rivals. Using reads to annotate the genome: influence of length, background distribution, and sequence errors on prediction capacity. *Nucleic Acids Research*, page gkp492, 2009.

51. Sven Rahmann. Fast large scale oligonucleotide selection using the longest common factor approach. *J. Bioinformatics and Computational Biology*, 1(2):343–362, 2003.

52. J. Rautio, J. Tanninen, and J. Tarhio. String matching with stopper encoding and code splitting In: Proc. CPM '02, Combinatorial Pattern Matching (ed. A. Apostolico, M. Takeda), Lecture Notes in Computer Science 2373, Springer, 2002, 42–52.

53. Eric Rivals, Anthony Boureux, Mireille Lejeune, Florence Ottones, Oscar Pecharromn Prez, Jorma Tarhio, Fabien Pierrat, Florence Ruffle, Thrse Commes, and Jacques Marti. Transcriptome Annotation using Tandem SAGE Tags. *Nucleic Acids Res.*, 35(17):e108, 2007.

54. Eric Rivals, Leena Salmela, Petri Kalsi, Petteri Kiiskinen, and Jorma Tarhio. Mpscan: fast localisation of multiple reads in genomes. In S. Salzberg and T. Warnow, editors, *Proc. 9th Workshop on Algorithms in Bioinformatics (WABI'09)*, Lecture Notes in Computer Science, Philadelphia, Sept 2009. Springer-Verlag.

55. L. Salmela, J. Tarhio, and J. Kytöjoki. Multipattern string matching with q-grams. *ACM Journal of Experimental Algorithmics*, 11(1.1):1–19, 2006.

56. S.S. Sheik, S.K. Aggarwal, A. Poddar, N. Balakrishnan, and K. Sekar. A FAST pattern matching algorithm. *J. Chem. Inf. Comput. Sci.*, **44**(4):1251–1256, 2004.

57. Andrew Smith, Zhenyu Xuan, and Michael Zhang. Using quality scores and longer reads improves accuracy of solexa read mapping. *BMC Bioinformatics*, 9(1):128, 2008.

58. Temple F. Smith and Michael S. Waterman. Identification of common molecular subsequences. *J. of Molecular Biology*, 147:195–197, 1981.

59. Marc Sultan, Marcel H. Schulz, Hugues Richard, Alon Magen, Andreas Klingenhoff, Matthias Scherf, Martin Seifert, Tatjana Borodina, Aleksey Soldatov, Dmitri Parkhomchuk, Dominic Schmidt, Sean O'Keeffe, Stefan Haas, Martin Vingron, Hans Lehrach, and Marie-Laure Yaspo. A Global View of Gene Activity and Alternative Splicing by Deep Sequencing of the Human Transcriptome. *Science*, 321(5891):956–960, 2008.

60. D.M. Sunday. A very fast substring search algorithm. *Communications of the ACM*, **33**(8):132–142, 1990.

61. J. Tarhio and H. Peltola. String matching in the DNA alphabet. *Software: Practice and Experience*, **27**(7):851–861, 1997.

62. R. Thathoo, A. Virmani, S.Sai Lakshmi, N. Balakrishnan, and K. Sekar. TVSBS: A fast exact pattern matching algorithm for biological sequences. *Current Science* **91**(1):47–53, 2006.

63. S. Wu and U. Manber. A fast algorithm for multi-pattern searching, Report TR-94-17, Department of Computer Science, University of Arizona, Tucson, AZ, 1994.

64. Z. Zhang, S. Schwartz, L. Wagner, and W. Miller. A greedy algorithm for aligning DNA sequences. *J. of Computational Biology*, 7(1-2):203–214, 2000.

65. R.F. Zhu and T. Takaoka. On improving the average case of the Boyer–Moore string matching algorithm. *Journal of Information Processing*, **10**(3):173–177, 1987.