BIOINFORMATICS
Vol. 17 Suppl. 1 2001
Pages S207–S214

An algorithm for finding signals of unknown length in DNA sequences

Giulio Pavesi[1], *Giancarlo Mauri*[1] *and Graziano Pesole*[2]

[1]*Department of Computer Science, Systems and Communication, University of Milan–Bicocca, Via Bicocca degli Arcimboldi 8, Milan, I-20126, Italy and*
[2]*Department of General Physiology and Biochemistry, University of Milan, Via Celoria 20, Milan, I-20133, Italy*

Received on February 5, 2001; revised and accepted on April 3, 2001

ABSTRACT

Pattern discovery in unaligned DNA sequences is a challenging problem in both computer science and molecular biology. Several different methods and techniques have been proposed so far, but in most of the cases signals in DNA sequences are very complicated and avoid detection. Exact exhaustive methods can solve the problem only for short signals with a limited number of mutations. In this work, we extend exhaustive enumeration also to longer patterns. More in detail, the basic version of algorithm presented in this paper, given as input a set of sequences and an error ratio $\epsilon < 1$, finds all patterns that occur in at least q sequences of the set with at most ϵm mutations, where m is the length of the pattern. The only restriction is imposed on the location of mutations along the signal. That is, a valid occurrence of a pattern can present at most $\lceil \epsilon i \rceil$ mismatches in the first i nucleotides, and so on. However, we show how the algorithm can be used also when no assumption can be made on the position of mutations. In this case, it is also possible to have an estimate of the probability of finding a signal according to the signal length, the error ratio, and the input parameters. Finally, we discuss some significance measures that can be used to sort the patterns output by the algorithm.

Contact: pavesi@disco.unimib.it

INTRODUCTION

Signal finding (pattern discovery) in DNA sequences is one of the most challenging problems in both molecular biology and computer science. Indeed, such signals that correspond to oligonucleotide patterns significantly over–represented in a set of functionally equivalent sequences are good candidates of some functional activity in the regulation of gene expression. In its simplest form, the problem can be formulated as follows: given a set of sequences, find an unknown pattern that occurs in at least q sequences of the set. If a pattern m letters long appears exactly in every sequence, a simple enumeration of all m–letter patterns that appear in the sequences gives the solution.

Alas, things are not that easy with biological sequences: signals present mutations, insertions or deletions of nucleotides, and usually do not occur exactly. Thus, while approximate occurrences of a single pattern can be still found efficiently, searching for all the 4^m possible patterns becomes quickly time consuming and feasible for very small values of m (Staden, 1989; Pesole *et al.*, 1992; Wolferstetter *et al.*, 1996; Tompa, 1999), even if we allow only mutations. This fact has led to the introduction of different heuristics to prune the search space, either by searching only for a subset of all possible patterns (for example, those that occur exactly in the sequences at least once, as in (Li *et al.*, 1999) and (Gelfand *et al.*, 2000)), or by imposing restrictions on the location of mismatches along the pattern, as in (Brazma *et al.*, 1998) and (Califano, 2000), where mismatches can occur only at fixed positions.

However, these methods do not work in the most general (and perhaps common) version of the problem, where the signal may not show up exactly, and mutations can occur anywhere. Thus, in order to discover longer and subtler signals, in the most widely used algorithms the pattern driven approach has been abandoned, and the signal is extracted by analyzing and comparing the patterns that occur in the sequences (see for example (Lawrence *et al.*, 1993; Bailey & Elkan, 1995; Hertz & Stormo, 1999; Pevzner & Sze, 2000; Buhler & Tompa, 2001)).

The algorithm we present can be described as "almost" exact. That is, we start from an exact algorithm based on suffix trees, where the time needed to find the occurrences of a pattern depends on the number of its valid occurrences in the sequences, rather than the size of the sequences themselves. Then, instead of reducing the number of patterns to be searched, we speed up the algorithm by narrowing down the set of valid occurrences for each pattern. To achieve this, we impose a restriction on the

location of mismatches. More in detail, given a pattern $p = p_1 \dots p_m$ and an error ratio $\epsilon < 1$, a valid occurrence of p in a sequence $s = s_1 \dots s_n$ is a subsequence $s_{i+1} \dots s_{i+m}$ such that, for all $j \in \{1, \dots, m\}$, the number of mismatches between $p_1 \dots p_j$ and $s_{i+1} \dots s_{i+j}$ is at most $\lceil \epsilon j \rceil$. In other words, every valid occurrence of p must start at the position of a valid occurrence of all its prefixes. This also permits to implement an efficient recursive search method, where no pattern length has to be explicitly given as input. The downside is obvious: since we consider only a subset of all its possible occurrences, some significant signal that does not show up in the forms "seen" by the algorithm might be missed altogether. However, we will show how the algorithm can work also when no assumption is made on the location of mutations, keeping an acceptable running time. Moreover, it is always possible to compute explicitly a priori the probability for a signal to be missed, according to its characteristics (i.e. length, maximum number of mutations allowed), and the parameters given as input to the algorithm. Thus, the user can choose a parameter setting that represents an optimal time/accuracy trade–off. Finally, we show different methods that we adopted to rank the patterns reported by the algorithm according to their significance.

SUFFIX TREES

A *suffix tree* is a data structure that exposes the internal structure of a string in a very deep and meaningful way. A suffix tree $\mathcal{T}$ for an n–character string $S = s_1 \dots s_n$ is a rooted directed tree with exactly n leaves numbered 1 to n (Weiner, 1973). Each internal node, other than the root, has at least two children. Each edge is labeled with a nonempty substring of S. Two edges leaving the same node cannot have labels beginning with the same character. For any leaf i, the concatenation of the edge labels on the path from the root to leaf i exactly spells the suffix of S starting at position i, that is, it spells out $s_i \dots s_n$.

The same structure can be built also for a set of k sequences. The easiest way is to append a different marker symbol, not occurring elsewhere, to each sequence in the set, so to distinguish which sequence a suffix belongs to, and to add the sequences to the tree one by one (see Figure 1). Moreover, it is also possible to annotate each node of the tree with a k–bit string, where the i–th bit is set if the word spelled by the path ending at the node occurs in the i–th sequence. The construction of the structure requires $O(N)$ time and $O(N)$ space (Weiner, 1973; McCreight, 1976; Ukkonen, 1995), where N is the overall length of the sequences, while annotating it with the bit strings takes additional $O(kN)$ time. However, these theoretical time and space bounds prove themselves to be very efficient in practice, as we will discuss later. When

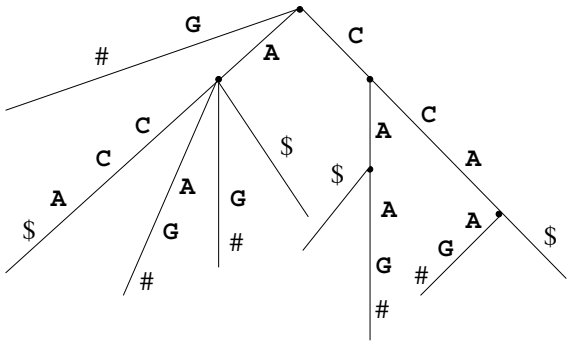

Fig. 1. Generalized suffix tree for sequences `ACCA` and `CCAAG`. Symbols $ and # are used as markers.

only exact pattern occurrences are taken into account, suffix trees seem to provide the most efficient solutions to pattern discovery problems, as shown in (Hui, 1992) and (Apostolico *et al.*, 2000). A thorough analysis of suffix trees, their virtues, and the different methods for their construction can be found in (Gusfield, 1997).

Given a set of sequences and the annotated suffix tree, *every* word appearing in at least one sequence of the set is spelled by a unique path starting from the root of the tree. Thus, searching for a pattern p in the set of sequences is straightforward. Starting from the root, we match the symbols of p along the unique path in the tree until either p is exhausted, or no more matches are possible. In the former case, the bit string on the next node on the path specifies which sequences p appears in.

We can also search for a pattern p with at most e mismatches in a similar way. In this case, we match p along different paths on the tree at the same time, keeping track of the number of mismatches encountered on each path. Whenever the number of errors on a path is greater than e, we discard that path. If we complete p, the surviving paths represent all the occurrences of p in the sequences with at most e mismatches. The sequences p appears in are given by the logical OR of the bit strings corresponding to the different paths.

THE WEEDER ALGORITHM

The starting point of our algorithm is the search method for approximate occurrences of a pattern outlined at the end of the previous section. Given a set of k sequences on the alphabet $\Sigma = \{\mathtt{A}, \mathtt{C}, \mathtt{G}, \mathtt{T}\}$, we want to find all (M, e) patterns, that is, patterns of length M that occur with at most e mutations in at least q sequences of the set. Let us suppose we have found on the tree the endpoints of paths corresponding to the occurrences of a pattern

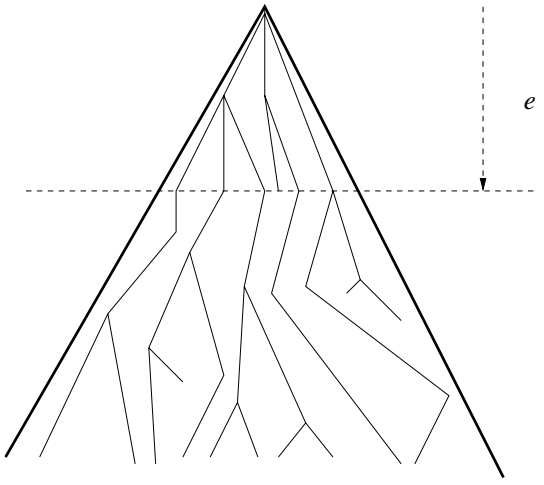

Fig. 2. Searching for (p, e) patterns with a suffix tree in the exact algorithm. At the beginning of the search, all paths of length e are valid.

$p = p_1 \ldots p_m$ in the sequences, that is, all the paths that spell words within distance e from p, with $m < M$. Also, we have associated with each path the distance (number of mismatches) from p of the corresponding substring. If p is valid, that is, occurs in at least q sequences (at least q bit are set in the resulting bit string), we try and expand it by one symbol. For each character $b \in \{\mathtt{A}, \mathtt{C}, \mathtt{G}, \mathtt{T}\}$, we match b against the next symbol on each path. If a path ends just before a node T of the tree, we match b against the first symbol on each edge leaving T. Whenever we encounter a mismatch, we increase the previous error along the path by one. Otherwise, the error remains unchanged. If the new error is greater than e, we discard the path. Once all paths have been checked, the surviving ones represent the approximate occurrences of $p' = p_1..p_m b$. If p' occurs in at least q sequences, and is shorter than M, we expand it as well; otherwise, we continue with p and the next character in Σ. The algorithm starts with the empty pattern from the root of the tree, and recursively expands it. That is, it matches the first symbol on each edge leaving the root against A. If A occurs in at least q sequences, it is expanded to AA. If also AA is valid, we move on to AAA, and so on. If it is not valid, we proceed with C, looking for occurrences of AC. Notice that in this method patterns do not have to occur exactly in the sequences.

This approach was first introduced in (Sagot, 1998) and (Marsan & Sagot, 2000), where it is described more in detail. For every valid pattern, we have to follow at most N different paths in the tree, where N is the overall length of the k sequences. For every word of length M spelled by a path in the tree (since the tree has N leaves, they are at most N), there are at most $\sum_{i=1}^{e} \binom{M}{i} (|\Sigma| - 1)^i \leq |\Sigma|^e M^e$ different patterns within distance e. The overall time complexity of the algorithm is thus $O(|\Sigma|^e M^e k N)$, where the additional k factor is needed to OR the bit strings. The algorithm is therefore exponential in the number of mutations allowed, and no longer in the length of the patterns. The main drawback, however, is that every pattern of length e satisfies the input constraints, since every other pattern of length e found in the tree is a valid occurrence for it (see Figure 2). That is, the algorithm has at least 4^e valid patterns of length e to expand, always starting with 4^e paths (or N, if $N < 4^e$). Thus, the method works well only for small values of e, like in the problem presented in (Tompa, 1999).

To apply the algorithm also to longer patterns with higher values of e one could reduce the number of patterns that have to be searched, for example those that occur exactly in the sequences, by checking that at least one path has error zero. The approach we chose, however, is different. Instead of reducing the set of patterns that have to be searched, we restrict the number of paths that have to be followed for each pattern. That is, we narrow down the set of valid occurrences. We will now show how.

Consider the following example. We want to find patterns of length 16 that occur with at most 4 errors. As previously explained, the search for each pattern will start with 4^4 paths. Among these paths, 3^4 will spell words at distance four from the pattern, and are thus very unlikely to lead to a valid occurrence. The idea is to "weed out" all these paths. We might also weed out paths with error three and two, considering only paths with at most one mismatch.

We implemented this method in the algorithm by determining dynamically the error threshold according to the pattern length. We fix an initial error ratio ϵ. Given a pattern p, a path is valid if the distance from p of the word spelled by the path is not greater than $\lceil \epsilon |p| \rceil$, where $|p|$ is the length of the pattern. If in the above example we set $\epsilon = 0.25$, we eliminate all paths of length four with error greater than one. When we expand p by one symbol, the error threshold is set to $\lceil \epsilon(|p| + 1) \rceil$. The result is that, for every pattern $p = p_1 \ldots p_m$, valid occurrences are words $s_{i+1} \ldots s_{i+m}$ occurring in the sequences for which:

$$\forall j \in \{1, \ldots, m\} \quad d(p_1..p_{1+j}, s_{i+1}..s_{i+j}) \leq \lceil \epsilon j \rceil$$

where $d(p_1..p_{1+j}, s_{i+1}..s_{i+j})$ is the number of mismatches between $p_1..p_{1+j}$ and $s_{i+1}..s_{i+j}$.

That is, $s_{i+1}..s_{i+m}$ is a valid occurrence for p if it is a valid occurrence for all its prefixes $\{p_1, p_1 p_2, \ldots, p_1 p_2 \ldots p_{m-1}\}$. In other words, we can see p as composed of $\lceil \epsilon m \rceil$ blocks (see Figure 3). The i–th block starts at position $\lceil \frac{1}{\epsilon} \cdot (i - 1) \rceil$ of the pattern. Every occurrence of p must present at most one mismatch in the first

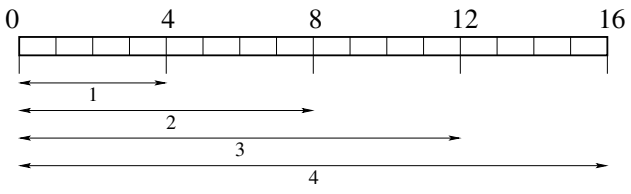

Fig. 3. Block decomposition of a pattern of length 16 with $\epsilon = 0.25$. At most one error is allowed in block one, at most two in blocks one and two, and so on.

block, at most two mismatches in the first two blocks, and so on. Now, the maximum number of paths that might correspond to a pattern of length m in the tree is $O(\lceil 1/\epsilon \rceil^{\epsilon m} |\Sigma|^{\epsilon m})$. The time complexity is thus reduced to $O(\lceil 1/\epsilon \rceil^{e} |\Sigma|^{e} kN)$, where M is the length of the longest pattern found, and $e = \lceil \epsilon M \rceil$. Since the error threshold is now determined dynamically, we no longer have to provide the algorithm with exact values for the pattern length and the maximum number of mutations. The core of the algorithm is the procedure **expand**, outlined in Figure 4.

It might seem that WEEDER can be applied only to a very limited subset of cases, where signals appear only in the forms considered by the algorithm. In the next section, however, we will show how it can be applied also to the general case of the problem, where no assumption is made on the location of mismatches, keeping a significant time improvement over the exact algorithm.

PERFORMANCE EVALUATION

In (Pevzner & Sze, 2000), the best currently available signal finding algorithms (namely, CONSENSUS (Hertz & Stormo, 1999), Gibbs sampler (Lawrence *et al.*, 1993), MEME (Bailey & Elkan, 1995), WINNOWER and SP–STAR (Pevzner & Sze, 2000)) are tested on the following problem:

Challenge Problem. *Find a signal in a sample of* 20 *sequences of fixed length, each containing an unknown pattern of length* 15 *with* 4 *mismatches*.

Sequences are sampled from i.i.d. sequences of fixed length, and a randomly mutated signal is implanted at randomly chosen positions in each sequence. The performance of the algorithms tested degrades as the length of the sequences is increased, and most of the methods fail to find the signal on sequences of length 600. On the other hand, both the length of the signals and the number of mutations are impractical for exact exhaustive methods. Thus, this can be considered a rather hard test for pattern discovery algorithms.

At first glance, the chances for WEEDER to find the signal seem pretty thin, since no assumption is made on the location of the mutations in its occurrences. A pattern of length 15 can appear with exactly four mutations in $\binom{15}{4} \cdot 3^4 = 1365 \cdot 3^4$ different ways. If we run the algorithm with $\epsilon = 0.26$, we have a block decomposition of $(3 - 4 - 4 - 4)$, and the number of patterns that are considered valid occurrences by the algorithm for a (15, 4) signal is $829 \cdot 3^4$. Therefore, the probability of finding the pattern in a single sequence is $p_{hit} = 0.61^{\dagger}$, and the probability of finding the pattern in all the 20 sequences is appallingly close to zero. On the other hand, running the algorithm in the exact version becomes time consuming even on limited sequence lengths.

Procedure **expand**(*pattern* p, Loc_p, *char* a)

1. $p' = pa$;
2. $errmax = \lceil \epsilon |p'| \rceil$;
3. $OccBits = [0, 0, ..., 0]$;
4. $Loc_{p'} = \emptyset$;
5. **for all** $(Pos, e) \in loc_p$
6. **for all** $Pos' \in Next(Pos)$
7. **if** $Last_{Pos'} = a$
8. $Loc_{p'} = Loc_{p'} \cup (Pos', e)$;
9. $OccBits = OccBits$ OR $Occ(Pos')$;
10. **else if** $e + 1 \leq errmax$
11. $Loc_{p'} = Loc_{p'} \cup (Pos', e + 1)$;
12. $OccBits = OccBits$ OR $Occ(Pos')$;
13. end **if**
14. end **if**
15. end **for**
16. end **for**
17. **if** (at least q bits are set in $OccBits$)
18. **for all** $a \in$ {A, C, G, T}
19. expand(p', $Loc_{p'}$, a);
20. end **for**
21. end **if**
22. return;

Fig. 4. The procedure **expand**. Letter a is appended at the end of pattern p. Loc_p is a set of pointers (Pos, e) to the endpoints of the occurrences of pattern p in the tree, with the corresponding error e; *OccBits* is a k–bit string representing the occurrences of p in the k strings; $Next(Pos)$ returns a set of pointers to the positions in the tree reached by moving by one letter down the path pointed by *Pos*; $Occ(pos)$ returns the bit string of the first node following the path pointed by *Pos*; $Last_{Pos'}$ is the last letter on the path ending at the position pointed by Pos'.

[†] If we consider also (15, 3), (15, 2) and (15, 1) patterns, we obtain a slightly higher success probability. However, the original experiment considered patterns that contained exactly four mutations.

However, we might try to use the algorithm as a sift. That is, we run it to find all patterns that occur in at least *half* of the sequences. This will take a bit longer, since the number of valid patterns is increased. This point will be discussed in the remainder of the paper. Then, we search the patterns reported by the algorithm, again using the suffix tree, this time with no restrictions on the location of the mismatches, in order to check which ones actually appear in all the sequences. We expect to perform this additional computation only on a limited number of candidates. Now, the probability that the algorithm finds the signal in the preliminary stage (possibly among other spurious random patterns) has risen to:

$$p_{hit}(20, 10) = \sum_{i=10}^{20} \binom{20}{i} p_{hit}^{i}(1 - p_{hit})^{(20-i)} = 0.89$$

If we lower the threshold to half minus one, the probability of finding the pattern becomes 0.95. We can also compute an estimate of the number of candidate patterns, that occur in at least half of the sequences and are passed by the algorithm to the second phase. Let

$$p_{(15,4)} = \sum_{i=0}^{4} \binom{15}{i} \left(\frac{3}{4}\right)^{i} \left(\frac{1}{4}\right)^{(15-i)}$$

be the probability that a pattern of length 15 occurs with up to four mutations in a given position of a random sequence. The probability that an occurrence of the pattern is "seen" by WEEDER is $p^{*}_{hit} = 0.63$ (higher than p_{hit}, since also occurrences with less that 4 mutations are considered). Thus, the probability that the pattern occurs in a position in a form valid for the algorithm is $p^{*}_{(15,4)} = p^{*}_{hit} \cdot p_{(15,4)}$, and that the pattern occurs at least once in a sequence of length n is $p^{seq}_{(15,4)} = 1 - (1 - p^{*}_{(15,4)})^{(n-14)}$. Finally, the probability that the pattern occurs in at least half of the sequences of the set and is found by the algorithm is estimated by:

$$p^{seq}_{(15,4)}(20, 10) = \sum_{i=10}^{20} \binom{20}{i} (p^{seq}_{(15,4)})^{i}(1 - p^{seq}_{(15,4)})^{(20-i)}$$

Thus, the expected number of patterns passed by the algorithm to the second phase can be estimated by $4^{15} \cdot p^{seq}_{(15,4)}(20, 10)$. In a set of twenty sequences of length 600 the latter is less than ten, while for sequences of length 1000 the number rises to about 300 patterns. When we look for patterns that occur in at least 9 sequences (in order to obtain a probability of success greater than 90%), the expected number of patterns passed to the second phase is about 50 for sequences of length 600 and 3500 for sequences of length 1000, values that can be easily handled by the algorithm.

Moreover, we could partition the set of sequences in two subsets of ten sequences each, and repeat the same procedure (at least half of the sequences must contain the pattern) on each subset. The probability that the pattern will pop up among the ones found by the algorithm in either subset is now

$$1 - (\sum_{i=6}^{10} \binom{10}{i} p_{miss}^{i}(1 - p_{miss})^{(10-i)})^2 = 0.98$$

where $p_{miss} = 1 - p_{hit}$. Notice that this can be performed with a single pass of the algorithm, by keeping two separate counters for the two subsets. A pattern is expanded whenever at least one of the two counters is greater than q, where q is a minimum number of sequences for each subset. Theoretically, this approach could be pushed even further. Partitioning the set of sequences, however, works as long as the parameters are such that only a few random patterns satisfy them, that is, the final exact search has to be performed on a limited number of patterns. This strategy works well on long signals, where the corresponding p_{hit} value is lowered by the block decomposition (like the (20, 6) signal shown in Table 1), and random patterns are unlikely to be picked up by the sifting phase. Thus, when the signal length and the number of mutations are known in advance, we can determine the best parameter setting and search strategy for WEEDER. The probability of finding a signal is not influenced by the nucleotide frequency, or by the length of the sequences. However, the parameters and the strategy have to be carefully chosen, since also random patterns could satisfy the input constraints in the sifting phase. This is the case, for example, of (14, 4) signals. The expected number of random patterns that satisfy the input constraints with $q = 10$ on twenty sequences of length 600 is in fact approximately 4500.

When the length of the pattern is not known in advance, there are some additional problems to be faced. The probability of finding a pattern depends on the block decomposition induced by ϵ. If we choose $\epsilon = 0.25$, the probability of hitting a (16, 4) pattern is 0.54, but the chances of finding a (15, 4) pattern have dropped to 0.45, since it is now decomposed as $(4 - 4 - 4 - 3)$. On the other hand, the probability of finding a (16, 4) pattern with $\epsilon = 0.26$ is 0.68, even if now we have to check whether the pattern we found actually occurs with four mutations or is a spurious (16, 5) pattern derived from a (15, 4) signal.

Moreover, setting a low threshold value q for the minimum number of sequences that have to contain an occurrence of a pattern increases the number of candidates that have to be searched in the second phase. If, for example, we run the algorithm with $q = 9$ and $\epsilon = 0.25$ on twenty sequences of length 400 hoping to find a (16, 4) pattern, the constraints are satisfied by hundreds

Table 1. Probability of finding a (m, e) signal with exactly e mutations in a single sequence (p_{hit}), and in an i.i.d. sample of 20 sequences according to the error ratio ϵ and the threshold q.

	(9, 3)	(10, 3)	(12, 3)	(14, 4)	(15, 4)	(16, 4)	(16, 5)	(20, 6)
(p_{hit}, ϵ)	(0.61, 0.33)	(0.74, 0.3)	(0.64, 0.25)	(0.60, 0.28)	(0.61, 0.26)	(0.54, 0.25)	(0.57, 0.31)	(0.48, 0.30)
$q = 10$	0.89	0.99	0.93	0.87	0.89	0.72	0.80	0.51
$q = 9$	0.95	–	0.97	0.94	0.95	0.84	0.90	0.58

of patterns of length twelve. One possible solution could be to investigate only the longest patterns found, but a significant signal could be hidden also among the shorter ones.

The solution we adopted is the following. We expand all patterns that appear in at least q sequences, but report only those that occur in $q(m)$, where $q(m)$ is set according to the pattern length. In the previous example, a pattern of length twelve with three mutations can be found with probability of about 90% by setting $q = 11$. Thus, a pattern of length twelve is passed to the second phase only if it appears in at least $q(m) = 11$ sequences. The $q(m)$ values can be set automatically by the algorithm according to the number of sequences given as input, and the parameters q and ϵ.

When the nucleotide composition of the sequences is not uniform, $q(m)$ should take into account also the probability of the pattern to occur. If we look for $(15, 4)$ patterns in a set of sequences with nucleotide frequency $1 : 1 : 1 : 2$ (that is, T occurs twice as often as the others), we might have hundreds of random patterns reported by the preliminary stage of the algorithm. So, in order to avoid spending too much time in the final phase we set the threshold $q(m)$ according to the pattern probability. The same idea can be applied also when the set of sequences is split. In any case, the points just discussed imply that at the end the algorithm might report more than one pattern satisfying the constraints. Hence the need to introduce significance measures to sort the output, as shown in the next section.

The same approach can be extended to corrupted samples (i.e. the signal appears only in a subset of the sequences), or to cases where patterns occur more than once in each sequence. In the latter, we just count the number of occurrences of the patterns without checking the bit strings. Moreover, when the pattern length is known in advance, an *ad hoc* block decomposition could be defined, in order to obtain better success probabilities while maintaining a significant improvement on the execution time with respect to the exact method.

SORTING THE OUTPUT

When the length of the signal to be found is unknown, or the sequences contain more that one signal, the algorithm usually outputs more than one pattern. This is the case, for example, of $(15, 4)$ patterns. When a successful $(15, 4)$ pattern is expanded by one symbol, it becomes a $(16, 5)$ pattern, and all its occurrences are also valid occurrences of the longer one. Thus, we added to the algorithm some methods for sorting the output patterns according to their significance. Measures of significance are computed by the algorithm during the final phase. Perhaps, given a pattern P, the simplest one is the following:

$$\mathcal{S}_P = |P| \cdot N_P - 2 \sum_{i=1}^{k} d(P, P_i) I(P, i)$$

where P_i is the best instance of P in sequence i, $I(P, i)$ is 1 if P appears in sequence i, zero otherwise, and N_P is the total number of sequences P occurs in. That is, we assign a premium score of $+1$ to every matching symbol, and a penalty of -1 to every mismatch. This measure works well to sort patterns that occur in all the sequences and when the probability distribution over the nucleotides is uniform. Moreover, the $d(P, P_i)$ values can be easily computed while searching for the pattern, by keeping an array of k elements corresponding to the minimum distance of P from each sequence. When the nucleotide composition of the sequences is biased, we use the background probabilities to define new match premiums and penalties. Analogously, we can use the *relative entropy*:

$$E_P = \sum_{j=1}^{m} \sum_{r \in \{A,C,G,T\}} p_{r,j} \log \frac{p_{r,j}}{b_r}$$

where $p_{r,j}$ is the frequency with which residue r occurs in position j in the occurrences of P, and b_r is the frequency of r in the sequences. This measure does not take into account the actual number of occurrences of the pattern. Thus, it is suitable to sort patterns that appear the same number of times, for example once in every sequence of the set, as in the case of sequences containing multiple signals. The observed number of occurrences of

P (denoted by N_P) can be added to the measure, by multiplying the relative entropy by $-N_P$. Also, we can define statistical measures of significance, that compare the actual number of occurrences of a pattern with the expected value:

$$Z_P = \frac{N_P - N\pi_{(P,e)}}{\sqrt{N\pi_{(P,e)}(1-\pi_{(P,e)})}}$$

where $\pi_{(P,e)}$ is the probability that P occurs in a sequence of the set with at most e mutations, and N is the overall length of the sequences. This value can be computed explicitly, as shown in (Tompa, 1999). However, we also adopted the *a posteriori* probability of P, that is, we compute π_P as the sum of the probabilities of the paths in the suffix tree corresponding to valid occurrences of P:

$$\tilde{\pi}_P = \sum_{P' \in \mathcal{O}(P,e)} \pi_{P'}$$

where $\mathcal{O}(P, e)$ is the set of patterns representing valid occurrences of P in the sequences, and $\pi_{P'}$ is the probability that P' appears exactly. Computing this value is straightforward with a suffix tree, and can also be used in the sifting stage to determine the threshold for patterns to be reported. Moreover, this measure implicitly considers how much the signal is conserved, since the more a pattern is conserved, the less paths will correspond to it in the suffix tree.

SOFTWARE IMPLEMENTATION AND EXPERIMENTS

The algorithm has been implemented on a Pentium II class computer with 64 Megabytes of RAM running the Linux operating system. The code has been written in ANSI C, and is about 2500 lines long. We tested it on the challenge problem previously described, varying the length of the sequences from 100 to 1000 nucleotides.

The construction of the generalized suffix tree always took less than one second. To save memory, we pruned the tree at a maximum path length of 100 symbols. Each node of the tree was annotated with the sequences k–bit string and a counter of the overall number of the occurrences of the corresponding word. The memory occupation, including the structures needed to save and rank the patterns, always remained under 10 Mbytes.

To find the (15, 4) signal with a 89% success probability, we ran the algorithm with $\epsilon = 0.26$ and $q = 10$. For sequence lengths up to 400 nucleotides, the algorithm took less than one minute (including the final exact search). The execution time started to grow significantly from length 500, since now more and more short random patterns satisfied the constraints. At lengths 500 and 600, the program took on the average 125 and 200 seconds to complete the execution. If we ran the algorithm with $q = 11$ (thus reducing the success probability to 78%), the time dropped to average values of 100 and 120 seconds.

On the other hand, increasing the number of sequences did not influence the execution time very much: for every sequence length, the algorithm took just a few more seconds when run on 30 or 40 sequences with q set to 16 and 21, respectively. Thus, WEEDER seems best suited to work on a very large set of relatively short (up to 600 nucleotides) sequences rather than a small set of very long sequences. In fact, when the sequence length was set to 800 nucleotides, the program took on the average 320 and 450 seconds to complete the execution, with q set to 11 and 10, respectively, while on length 1000 it took about fifteen minutes. The number of patterns searched in the final stage was always limited to a few hundreds, and this step took just a few additional seconds.

In a very recent work (Buhler & Tompa, 2001), a test similar to the one of (Pevzner & Sze, 2000) is performed on signals even harder to detect than the (15, 4) patterns of the Challenge Problem, that is, (14, 4), (16, 5), (18, 6), and (19, 6) implanted patterns in a set of 20 sequences of length 600. For the (14, 4) signal, running the algorithm with $q = 10$ and $\epsilon = 0.28$ took about ten minutes (since the increased value of ϵ also increased the number of short patterns that had to be expanded) and the patterns passed to the second phase were a few thousands. However, the final exact search took less than a minute to be completed. Instead, when the value for ϵ had to be greater than 0.3 (as in the other cases just mentioned) and the threshold q had to be lower that one half (given the low hit probability induced by the block decomposition) the time required by the algorithm was significantly increased, in some cases exceeding an hour, and a larger (hundred of thousands) number of candidate signals had to be examined in the second phase. Thus, the error ratio value of 0.3 seems to be a critical point for the efficiency of the algorithm.

The complete software implementation of WEEDER is in the beta testing stage, and we plan to release it free of charge in the near future. Moreover, we are currently developing a parallel version of the algorithm, that runs on a cluster of workstations.

CONCLUSIONS

We have presented an algorithm for pattern discovery in DNA sequences, that permits to extend exhaustive enumeration to signals longer than those usually considered by exact methods, and does not need as input the exact length of the patterns to be found. The speedup has been obtained by imposing restrictions on the positions where mutations can occur. However, we have shown how the algorithm can be used also when no assumption can be made on the location of mismatches, by applying it to a

rather hard benchmark problem. Patterns that satisfy the input constraints can be sorted according to different significance measures. Quite naturally, the next step in the evaluation of the performance of the method will be its application to real biological instances. A software package based on the algorithm will be released in the near future. Moreover, we plan to extend the algorithm in order to deal with gapped signals, and to work also on amino acid sequences.

ACKNOWLEDGEMENTS

This work has been supported by the Italian Ministry of University, under the project "Bioinformatics and Genomic Research".

REFERENCES

Apostolico, A., Bock, M., Lonardi, S. & Xu, X. (2000). Efficient detection of unusual words. *Journal of Computational Biology*, **7**, 71–94.

Bailey, T. & Elkan, C. (1995). Unsupervised learning of multiple motifs in biopolymers using expectation maximization. *Machine Learning*, **21**, 51–80.

Brazma, A., Jonassen, I., Vilo, J. & Ukkonen, E. (1998). Predicting gene regulatory elements in silico on a genomic scale. *Genome Research*, **8**, 1202–1215.

Buhler, J. & Tompa, M. (2001). Finding motifs using random projections. In *Proceedings of the Fifth Annual International Conference on Computational Molecular Biology (RECOMB 2001)*. To appear.

Califano, A. (2000). Splash: structural pattern localization analysis by sequential histograms. *Bioinformatics*, **16**, 341–357.

Gelfand, M., Koonin, E. & A.Mironov (2000). Prediction of transcription regulatory sites in arachea by a comparative genomic approach. *Nucleic Acids Research*, **28**, 695–705.

Gusfield, D. (1997). Algorithms on Strings, Trees and Sequences: Computer Science and Computational Biology. Cambridge University Press, New York.

Hertz, G. & Stormo, G. (1999). Identifying dna and protein patterns with statistically significant alignment of multiple sequences. *Bioinformatics*, **15**, 563–577.

Hui, L. (1992). Color set size with applications to string matching. In *Proc. 3rd Symp. on Combinatorial Pattern Matching, Springer Verlag LNCS 644*. pp. 227–240.

Lawrence, C., Altschul, S., Boguski, M., Liu, J., Neuwald, A. & Wooton, J. (1993). Detecting subtle sequence signals: a gibbs sampling strategy for multiple alignment. *Science*, **262**, 208–214.

Li, M., Ma, B. & Wang, L. (1999). Finding similar regions in many strings. In *Proceedings of the 31st Annual ACM Symposium on Theory of Computing*. pp. 473–482.

Marsan, L. & Sagot, M. (2000). Algorithms for extracting structured motifs using a suffix tree with application to promoter and regulatory site consensus identification. *Journal of Computational Biology*, **7**, 345–360.

McCreight, E. (1976). A space-economical suffix tree construction algorithm. *Journal of the ACM*, **23**, 262–272.

Pesole, G., Prunella, N., Liuni, S., Attimonelli, M. & Saccone, C. (1992). Wordup: an efficient algorithm for discovering statistically significant patterns in dna sequences. *Nucleic Acids Research*, **20**, 2871–2875.

Pevzner, P. & Sze, S.-H. (2000). Combinatorial approaches to finding subtle signals in dna sequences. In *Proceedings of the Eighth International Conference on Intelligent Systems for Molecular Biology*.

Sagot, M. (1998). Spelling approximate repeated or common motifs using a suffix tree. *Lecture Notes in Computer Science*, **1380**, 111–127.

Staden, R. (1989). Methods for discovering novel motifs in nucleic acid sequences. *Computer Applications in Biosciences*, **5**, 293–298.

Tompa, M. (1999). An exact method for finding short motifs in sequences, with application to the ribosome binding site problem. In *Proceedings of the Seventh International Conference on Intelligent Systems for Molecular Biology*.

Ukkonen, E. (1995). On-line construction of suffix trees. *Algorithmica*, **14**, 249–260.

Weiner, P. (1973). Linear pattern matching algorithms. In *Proc. of the 14th IEEE Symp. on Switching and Automata Theory*. pp. 1–11.

Wolferstetter, F., Frech, K., Herrmann, G. & Werner, T. (1996). Identification of functional elements in unaligned nucleic acids sequences by a novel tuple search algorithm. *Computer Applications in Biosciences*, **12**, 71–80.