A High Performance Distributed Tool for Mining Patterns in Biological Sequences

Pedro Pereira, Nuno A. Fonseca, Fernando Silva

Technical Report Series: DCC-06-08

Departamento de Ciência de Computadores
&
Laboratório de Inteligência Artificial e Ciência de Computadores
Faculdade de Ciências da Universidade do Porto
Rua do Campo Alegre, 1021/1055,
4169-007 PORTO,
PORTUGAL
Tel: 220 402 900 Fax: 220 402 950
http://www.dcc.fc.up.pt/Pubs/

A High Performance Distributed Tool for Mining Patterns in Biological Sequences

Pedro Pereira*, Nuno A. Fonseca, Fernando Silva

{pdr,nf,fds}@dcc.fc.up.pt
DCC - FC & LIACC, Universidade do Porto
Rua do Campo Alegre, 1021/1055, 4169 - 007 Porto – Portugal

Abstract

The identification of interesting patterns (or subsequences) in biosequences has an important role in computational biology. Databases of genomic and proteomic sequences have grown exponentially, and therefore pattern discovery is a hard problem requiring clever strategies and powerful pattern languages to achieve manageable levels of efficiency. As far as we are aware of, known tools are either inefficient or can only handle limited amounts of data. We present a new pattern discovery tool called BIORED for mining patterns in proteomic and genomic sequences. It uses a genetic algorithm to find interesting patterns in the form of regular expressions, and a new efficient pattern matching procedure to count pattern occurrences. The counting operation is still the bottleneck of the matching algorithm, therefore we propose a parallel and distributed version of this algorithm. It adequately partitions the sequence being searched among the processing units so that these can perform in parallel, and independently, the counting for a set of occurring patterns. We validate the tool's usefulness and study the sequential and parallel performance of the base algorithm on several data sets. Our results indicate that BIORED scales well, achieving almost linear speedups, up to 22 processors, on a distributed memory computer.

1 Introduction

In the last years, the amount of biotech data has increased exponentially, namely in genomic [3] and proteomic [12] data. The volume of data has been doubling every three to six months as a result of automation in biochemistry and projects of genome sequencing. Processing all data is either impossible or very expensive, both humanly and computationally. Thus, automatic analysis of the large quantities of data is important.

A pattern, or sequence motif, is a repeating subsequence. Patterns often have an important biological significance, hence pattern discovery is an important problem in computational biology. It is also a computational hard problem since the combinatorial involved is extremely large.

The rationale behind pattern discovery in biosequences (proteomic and genomic) is that the patterns correspond to subsequences preserved through evolution, and the reason for being preserved is that they are important to the function or structure of the molecule. For instance, non-coding genetic sequences are generally not well preserved, thus the presence of a conserved sequence (or a pattern) usually implies that it is functional important. Patterns in non-coding regions can help to determine the function of nucleotide sequences (see e.g., [7]).

Several pattern discovery programs in biosequences have been proposed in the literature in the past years, such as Pratt [14], MEME [2] and TEIRESIAS [21], just to mention a few. The types of patterns that these tools are able to handle range from simple strings to quite general regular expressions.

*Corresponding author.

$$
\begin{array}{lcl}
< pattern > & ::= & < pattern >< position >|< empty > \\
< position > & ::= & < symbol >| \, . \\
< position > & ::= & [< symbol - list >] \\
< position > & ::= & [\, \hat{} < symbol - list >] \\
< symbol - list > & ::= & < symbol - list >< symbol >|< empty > \\
< empty > & ::= & \epsilon
\end{array}
$$

Figure 1: Pattern language. A *symbol* is any element belonging to Σ.

Many proposed approaches generate all possible patterns and then verify, for each one, its support. However, many of the pattern discovery programs in biosequences are not freely available, and those available usually use too much memory and/or are too slow in their execution.

In this paper we describe BIORED, a tool to discover patterns in genomic and proteomic sequences. It accepts a powerful pattern language that is a subset of regular expressions. We use a genetic algorithm to discover patterns together with an efficient pattern matching procedure to count pattern occurrences in the sequences. To achieve higher performances we propose a parallel and distributed version of BIORED. In this version, the sequence being searched is partitioned among the processing units so that they can perform in parallel, and independently, the counting of the number of occurrences for a set of patterns occurring in the sequence. We validate the usefulness of BIORED by applying it to several datasets and studying its sequential and parallel performance. Our parallel results show that we achieve high scalability on distributed memory computers.

The contributions of this paper are two-fold. First, we describe and evaluate a genetic algorithm to find patterns in genomic and proteomic sequences that uses an efficient pattern matching procedure, a crucial component for achieving high performance in any pattern discovery tool. Second, for a data mining practitioner, we propose an efficient and useful tool for mining patterns in biosequences.

The remainder of the paper is organized as follows. We next describe the pattern language and define the problem addressed. In Section 3 is described the genetic algorithm for pattern discovery, together with a sequential and a parallel matching procedures. The BIORED is validated in Section 4, followed by a performance study. In Section 6 our proposal is related with relevant previous work. We conclude with some open research problems in Section 7.

2 Preliminaries

The problem of pattern discovery here addressed can be stated as follows. Let Σ be an alphabet of residues (proteomic or genomic). Given a set of sequences S, each sequence composed with characters not restricted to the alphabet Σ, and a pattern size (k), the goal is to find the best interesting pattern (p) with size k accordingly to some evaluation function.

We consider deterministic patterns with wild-cards and ambiguous characters. More specifically, the pattern language is a subset of regular expressions. Every position in the regular expression can only be composed by classes of characters belonging to Σ. A class is represented within brackets. The "." (referred to as don't care character) is used to denote a class of characters composed by all elements in Σ. For compactness of representation, it is also possible to negate the class. In this case, all characters belonging to the alphabet not present are the ones that compose the class. The negation is denoted by "^". The patterns are described by the BNF grammar in Figure 1. The non-terminal `symbol` varies accordingly to the alphabet used.

A few examples of patterns (for $\Sigma = \{G, T, C, A\}$) follows:

- `ATGACAGTA`
- `A[AC]GT[ACGT]`

- `[AC][GT][AT][AGT][ACGT][TG]`
- `[^A]GTTG[^T][^C]`
- `GT.TG[^C]`

A pattern p being a regular expression defines a language $L(p)$. The elements of the language are all the subsequences (substrings or words) that can be obtained by p. The "." appearing in a pattern can be substituted by an arbitrary element of Σ. A class of characters appearing in a pattern can be substituted by any element of Σ appearing withing brackets. For instance, the $L(A[GT].) = \{AGA, AGC, AGG, AGT, ATA, ATC, ATG, ATT\}$.

A pattern p is said to have a match in a sequence if the sequence contains at least one word that belongs to $L(p)$. For instance, the pattern with length 3 "`[GT].A`" has two matches in the sequence $AA\underline{TAA}G\underline{TTA}A$.

The chosen pattern language is a compromise between simplicity and power. The idea is to allow the discovery of complex patterns while having a sufficiently fast matching algorithm. Although interesting patterns may have gaps, which may be the result of deletions or insertions, many others have underwent smaller mutations and have an equal length. The principle is that we can usually find sub-patterns of larger patterns and later extend them.

Another advantage of using (a subset) of regular expressions results of the language being well supported by a considerable number of programs (e.g., grep, sed, emacs, etc) and programming languages (e.g., Perl, PHP, etc).

3 A Genetic Algorithm for Pattern Discovery

A genetic algorithm (GA) [16] is a search strategy that use techniques inspired by evolutionary biology such as inheritance, mutation, selection, and crossover (also called recombination) to find approximate solutions to optimization and search problems. They are a good option to tackle the problem of pattern discovery as we are interested in finding good enough patterns and not only the best one.

A GA is typically implemented as a computer simulation in which a population of abstract representations (called chromosomes) of candidate solutions (called individuals) evolves towards better solutions. The evolution starts from an initial population of individuals and happens in generations. In each generation, the fitness of the whole population is evaluated, multiple individuals are randomly selected from the current population (based on their fitness), and modified (mutated or recombined) to form a new population. The new population is then used in the next iteration of the algorithm. This generational process is repeated until a termination condition is reached (e.g., threshold in the number of generations).

The implementation of a GA requires the prior definition of a (1) a genetic representation of a pattern (solution), and (2) a fitness function to evaluate the patterns. The implementation of the fitness function involves counting the number of matches of a pattern in the input sequences, which is, naturally, an important factor in the speed and efficiency of the algorithm. We therefore devised an efficient matching procedure, and later, parallelized it.

We propose a genetic algorithm to perform pattern discovery that receives as input a set of sequences, the length of a pattern (k), and some other parameters (such as the maximum number of generations i), and tries to find an interesting pattern of length k. We next describe its implementation, namely the representation of the individuals (patterns), the initial population (set of patterns), the genetic operators used, the fitness function and the sequential and parallel algorithm for counting the matches. We end with some notes regarding the implementation.

3.1 Representation of the Individuals

We use a binary vector as a chromosome to represent a pattern. The binary vector can, conceptually, be seen as signaling if a character belonging to Σ is present or not in a determinate pattern position.

For example, the DNA pattern `[AC]T[ACGT]G` is represented as `1100,0001,1111,0010`, if `A` is represented with the bit-mask `1000`, `C` `0100`, `G` `0010` and `T` `0001`.

3.2 Initial Population

The initial population is randomly initialized. Each bit in an individual has the probability 0.7 of being activated (this value was selected after performing several experiments). The probability value is quite high because it is important that the initial patterns have several occurrences on the text. Setting it higher is problematic since the population diversity greatly decreases.

3.3 Fitness Function based on statistical significance

To guide the search for a pattern and for ranking a set of patterns one needs some measure to assess, in some way, their quality. In a GA context, such measure is called fitness function. In complex problems, such as pattern discovery, GAs have a tendency to converge towards local optima rather than the global optimum of the problem. This problem may be alleviated by using a different fitness function, or by using techniques to maintain a diverse population of solutions. Therefore, two fitness functions were considered based on statistical interestingness.

Several approaches have been proposed to determine if a pattern is statistically interesting [24, 23, 22], i.e., if the number of occurrences of a pattern in a set of sequences is greater or lower than the expected value. A pattern is considered statistically interesting if it is overrepresented in the sequences where it occurs. To measure the over-representation, we need to consider the expected number of occurrences and the standard deviation of this value. Equivalently, we need to know how the values are distributed.

We assumed that the probability of the symbols (from Σ) to appear in S are independent and identically distributed. Under these assumptions, the word probability follows a Binomial distribution. The Binomial distribution gives the discrete probability $b(x;\ n, p)$ of obtaining exactly x successes (matches) out of n Bernoulli trials (pattern positions). We consider every character position, that can be a possible place for the word occurrence, as a Bernoulli trial. For example, if we have the sequence `ACGATCAGTACA` and the pattern that we are computing the statistics for has length 5 then there are exactly 8 places where the pattern can occur. Generalizing, having a sequence and a word of length S_n and W_n respectively, there are $S_n - W_n + 1$ places where the word can appear if $S_n \geq W_n$ or zero otherwise. Each Bernoulli trial is true with probability p. The probability p is the multiplication of the probabilities of the individual pattern positions. In turn, the pattern positions probabilities is the sum of the probabilities of the symbols that compose the position. For efficiency reasons, the binomial distribution is approximated by the Poisson distribution for large values of n and small values of p, with $\lambda = np$, or equivalently $p(x;\ \lambda) \approx b(x;\ n, \lambda/n)$ [8].

We are interested to know if the pattern is overrepresented, therefore we calculate the probability of the pattern appearing at least the same number of times in the dataset than it did. Equivalently, we compute the complementary cumulative distribution function (F_c) of the Poisson distribution for $x-1$: $Z = F_c(x-1) = P(X > x-1)$. Since, the Z can take very small values we use the negative logarithmic of Z, more specifically, $-\log(Z)$. We next denote $-\log(Z)$ as $\mathcal{I}$.

The first fitness function relates the interestingness of the pattern with its complexity,

$$f_1 = \frac{\mathcal{I}}{\text{complexity}^x}$$

for $x = 0, 1, 2, 3$. The *complexity* is the sum of the number of characters recognized by each pattern position. For instance, `ACGT` has complexity 4, while `[AC]CGT` has complexity 5 and `[AC][CG][GT][TA]` has complexity 8. The parameter x is used to reduce the patterns complexity, thus improving their quality. Generally, the low quality patterns are a direct result of being too general.

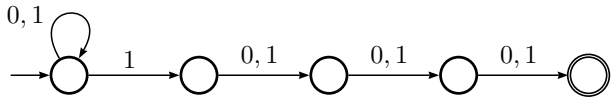


Figure 2: Exponential NFA to DFA construction case.

The other fitness function (f_2) borrows ideas from the evaluation function F-measure,

$$f_2 = \frac{2 \times \text{logpn} \times \text{cpx}}{\text{logpn} + \text{cpx}}$$

where p is the probability of the pattern, m the maximum complexity with same length and alphabet can have, $\text{cpx} = 1 - \frac{\text{complexity}}{m}$, $\text{logpn} = \frac{\mathcal{I}}{10000}$. A ceiling of 10000 is assumed to the value of $\mathcal{I}$.

In general, it is not possible to determine which fitness function behaves better in a set of sequences without some kind of experimentation. This experimentation needs only to be done once for each sequence, and can be done automatically by executing the programs for all the possible fitness functions and choosing the one that achieves the best results.

3.4 Counting Matches

The fitness function requires knowing the number of (overlapping) occurrences of every pattern in the sequence. For example, in the sequence `AAATAAA`, the pattern `AA` occurs four times and the pattern `AA[AT]` occurs three times.

Counting the number of occurrences of a single pattern can be troublesome. For instance, if the sequences have total length of n and the pattern is composed by either symbols or unit-length don't care characters (".") with length m, the best algorithm runs in $O(n \log m)$ time (worst-case) [6]. If we could come up with an algorithm that fast for the worst-case, the best we could do would be $O(ni \log m)$, where i is the number of different patterns (number of individuals of the population). However, since unit-length don't care characters are a subset of classes of characters, the chosen pattern language is more powerful than the pattern language referred in [6].

Since the GA generates several individuals (patterns) in each generation, that need to be evaluated, one try to devise a form to evaluate them simultaneously in a efficient manner.

A first solution for a (almost) linear patterns occurrence counter, is to use a finite state transducer (FST). A FST is a finite state machine (FSM) with two tapes (a finite state machine has only one), one for input and the other for output. The output would be the indexes of the patterns recognized (one for each occurrence). This appears to be a really good solution until we try to construct such a transducer. In FSMs there are cases where their construction can be exponential in the number of states, see Figure 2 for an example. One could think that the exponential case is very rare, and it would be right for a great number of regular expressions, which exclude the ones in our pattern language. The exponential worst case occurs exactly with the kind of regular expressions we consider, and thus, constructing the FST can not be done unless the pattern length and population number are small. If that is the case we do not really need the FSTs.

We decided to develop our own algorithm because we did not find in the literature [19, 11] a single one with the desired characteristics: fast and easy to implement.

If the algorithm can only handle a single pattern, then it is possible to use a linear solution based on bit-parallelism [18] if the pattern length is small (only a few machine words are needed). The bits are used to simulate a non-deterministic finite automaton (NFA) that describes the pattern. If we have the set of patterns `[AC][CG]C`, `ATG` and `A[CT]C`, we can see them as a NFA similar to the one plotted in Figure 3.

To expand the algorithm to evaluate several patterns at once, a window with length k is moved through the sequence. Note that all patterns have the same length k. For each window position

every pattern is checked for a match. In a sequence with size n, the number of window positions (window size is k) is $n - k + 1$ (assuming that $n \geq k$).

After representing the patterns as a NFA (for an arbitrary window position) we used bit-parallelism [18] for checking matches. The idea is to perform the operation at every window position as fast as possible by partition the patterns in groups of machine word bit-width, and iterate over the window size for each one of these groups. See Algorithm 1 for the algorithm description.

Algorithm 1 Match procedure.

```
input: The Pattern structure, the patterns length,
       number of patterns (size), and the size of
       the alphabet (symbols)
 1 ceil:=CeilDivide(size,MachineWordSize)
 2 SetZero(Mask)
 3 for i:=0 to symbols-1 do
 4   for j:=0 to length-1 do
 5     for k:=0 to size-1 do
 6       if IsSet(Pattern[k].Position[j].Symbol[i]) then
 7         Mask[i,j,k/MachineWordSize] |= 1 << k%MachineWordSize
 8       fi
 9     od
10   od
11 od
12 foreach (Window Position) do
13   for i:=0 to ceil-1 then
14     n:=AllBitsSet()
15     for j:=0 to length-1 do
16       n:=n & Mask[Window[j],j,i]
17       if n = 0 then break fi
18     od
19     if n <> 0 then
20       v:=GetSetBitIndex(n)
21       counter[v]:=counter[v]+1
22       UnsetBit(v,n)
23     fi
24   od
25 od
```

The counting matches algorithm worst-case complexity is $O(nik)$ with the input size n, i the number of individuals in the population, and k the length of the patterns. However, the algorithm is in average much faster, achieving a complexity of $O(ni/w)$, where w is the number of bits in a machine word. The average complexity is directly linked to the average case of the naive string matching.

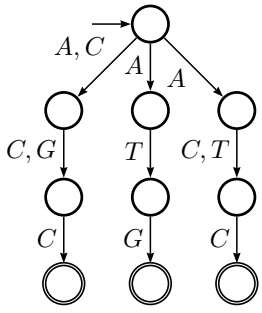


Figure 3: NFA for a pattern set.

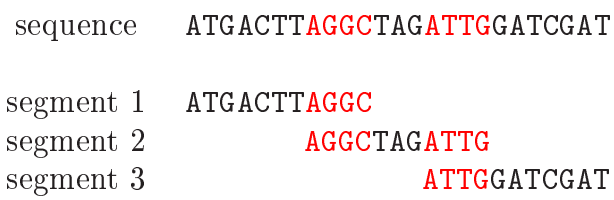


Figure 4: Data parallelism with size=25 and window=5 processors=3.

3.5 Counting Matches in Parallel

The counting operation is in fact the bottleneck of the matching algorithm, therefore one can expect to achieve major gains in execution speedup by doing it in parallel. This operation is highly amenable to data parallelism as the sequence being searched can be divided adequately among the processing units so that these can perform in parallel and independently the counting of the number of occurrences for a set of patterns occurring in the dataset. With this strategy we expect to achieve very low communication overheads and high scalability.

We divide the sequence among the available processors. Each processor maintains counters for all patterns that compose the population. These counters are merged with all the others, resulting in the total number of occurrences for each pattern.

We can not just divide the sequence in non-overlapping segments. If we did it, we could end up missing some occurrences that start in one segment and end in another segment. So, the segments had to overlap each other. We were, however, interested in having the least overlapping as possible, to reduce computation. Consider w as the pattern length. Obviously, no more than $w-1$ characters can be overlapped in two segments, as otherwise one occurrence of a pattern could be mistakenly counted twice. The partition scheme we developed is illustrated in Figure 4. If we have p processors available, the idea is to have $p-1$ overlapping segments of length $w-1$.

Our parallel algorithm is described in Algorithm 2. It is SPMD (Single Program Multiple Data) in which a "master" process is responsible for initially distributing the sets of patterns and the "master" as well as the other "worker" processes compute the counters in parallel. For every set of patterns we initialize the pattern occurrence counters. The pattern information is broadcasted to all processes. Each process executes the matching algorithm on its data segment to determine the local occurrence counters, and then contributes with its counters to a global add operation to determine the total number of occurrences for the patterns.

3.6 Genetic Operators

A genetic operator [16] is a process that aims at maintaining genetic diversity. The operators are analogous to those that occur in the natural world: survival of the fittest, or selection; sexual or asexual reproduction, or crossover; and mutation.

We implemented and evaluated two selection operators. The first one was fitness proportionate [16], the most used selection operator in GAs. It lead to poor results because no upper bound on the evaluation function could be computed; if the fitness of the fittest individual was much higher than the average fitness the search would narrow too quickly, leading to premature convergence and then to stagnation.

Next, we then implemented a rank selection operator that solved the referred problems. In rank selection the individuals in the population are sorted by comparing their fitness value. Each individual is then given a probability of being chosen for reproduction depending on its position. For n individuals, a $n(n+1)/2$ slots roulette-wheel is constructed, with the fittest individual receiving n slots, the second fittest $n-1$, and so on until, with least fit individual receiving just one slot.

During the alternation (or reproduction) phase of the GA, we use three classical genetic operators: mutation, crossover and elitism.

Algorithm 2 Parallel Match.

```
input: An array with sets of Patterns and the
       Dataset
 1 MpiInit()
 2 Rank := MpiRank()
 3 Size := MpiSize()
 4
 5 if (Rank = 0) then
 6   Length := ComputeDatasetLength(Dataset)
 7 fi
 8 Length := MpiBroadcast(Length)
 9
10 u := LeftInterval(Rank,Size,Length)
11 v := RightInterval(Rank,Size,Length)
12
13 N := size(Patterns)
14 for i := 0 to N-1 do
15   Initialize(Counters)
16   Patterns := MpiBroadcast(Patterns[i])
17   Counters := AlgorithmMatch(Dataset, Patterns[i], Counters,u,v)
18   Counters := MpiReduceAdd(Counters)
19 od
20
21 MpiShutdown()
```

The crossover operator selects a character position in the individual to be generated. It then sets the first part with the contents of the first individual and the second part with the contents of the second individual (both selected using a rank selection operator). That is, if u and v are the individuals chosen for crossover generating a new individual w, or,

$$u = u_1u_2u_3 \cdots u_k,\ v = v_1v_2v_3 \cdots v_k$$

would generate the following individual, if the crossover point was i,

$$w = u_1u_2 \cdots u_iv_{i+1}v_{i+2} \cdots v_k$$

where the u_i and v_i for $1 \leq i \leq k$ are classes of characters.

The mutation operator randomly flips some of the bits that compose the chromosome. The elitism operator selects some of the best individuals to be copied verbatim to the next generation, not suffering any mutation.

3.7 Implementation

The sequential and parallel GAs for pattern discovery were implemented using the C language because the speed was crucial and we needed extreme control about memory usage. For the statistic functions we used the R [20] library. For the parallel algorithm we used a reduced set of LAM functions, a MPI programming environment [5], thus making the implementation extremely portable between MPI environments.

The alphabet letters (representing nucleotides or aminoacids) are implemented using an unsigned integer with 32 bits. This representation has the advantage of being simple to apply the genetic operators, namely the crossover and the mutation. This means that a population with i individuals, each having length k, uses exactly $4ik$ bytes of memory using the DNA alphabet. In general, the algorithm uses $|\Sigma|ik/8$ bytes of memory, where Σ is the alphabet used.

The counters for the number of occurrences were implemented using 64-bit "doubles". They provide a 56-bit mantissa and are more portable than any other C type.

The probability of undergoing crossover was set to 0.75 and the mutation probability to 0.01. Only the fittest individual is considered an elite. These values were chosen after some experiments

with DNA and aminoacid sequences and are the values that proved to work better, in most times. By default, the program halts after completing 500 generations. This value was chosen based on the performance experiments presented in Section 6.

A final note. It is possible to use symbol probabilities (distribution) other than the observed in sequence given as input by providing an extra file to the program. This is useful because we might have a sequence slice with slightly skewed character probabilities, where there is nothing particularly interesting if the probabilities are gathered from the slice, but are highly interesting if the whole sequence probabilities are considered.

4 Validation

In this section we demonstrate the usefulness of the BIORED by applying it to several datasets (sets of sequences). The goal was to try to rediscover some already known patterns. We validate the patterns found by the genetic algorithm against known motifs and patterns referred in the literature. For the new patterns found we performed a BLAST [1] search to confirm that the program really finds statistically interesting patterns.

4.0.1 Human Gene for Proinsulin

The program was configured with a population of 32 individuals, pattern length of 14, and to stop after one-thousand generations. It yielded the pattern:

```
[CG][AT]GGGG[AT][CG][AT]GGGG[AT]
```

with a score of 381.6, occurring 48 times and with a probability of 0.00000133. The pattern found is very similar to a previously reported pattern `ACAGGGG TGTGGGG` [15].

4.0.2 Alpha Helix

We searched for interesting patterns in the beginning of alpha helices. The helices were obtained from a set of 835 protein chains with low homology. The Pisces server [25] was used to select the subset of 835 protein chains from the Protein Data Bank-PDB [4] with structures solved at a resolution higher than 1.6, with a R-factor lower than 25%, and showing a maximum of 20% homology.

The input given for the program was composed by 4479 sequences, each having the first four symbols of an alpha-helix. The program was executed by computing the symbol statistics from the the whole alpha sequences and not just the alpha helices start. The program was ran with a population size of 32 individuals, a pattern length of 4 and for 1000 generations.

The best pattern found was

```
[^CDHKNQSWY][DNPST][ADEKLPQV][ADEQ]
```

The pattern occurs 564 times, has a probability of 0.02193826 and a score of 1172.6. The pattern is consistent with the information provided in [9].

4.0.3 Drosophila Melanogaster

Drosophila melanogaster is a fruit fly, a little insect, of the kind that accumulates around spoiled fruit. It is one of the most valuable organisms in biological research, particularly in genetics because of the short life span and because the entire genome has been sequenced.

We started with a known consensus described in [7]. The consensus was manually converted to a regular expression (after some simplification), producing the following pattern:

```
[^G][AG]AGTT[CT]GT[^A][GT]C[CT]T[AG]AGTCTTT[CT]GTTT
```

The pattern achieved a score of 904.19 on the entire genome, i.e., the entire genome was used to compute the distribution of the symbols.

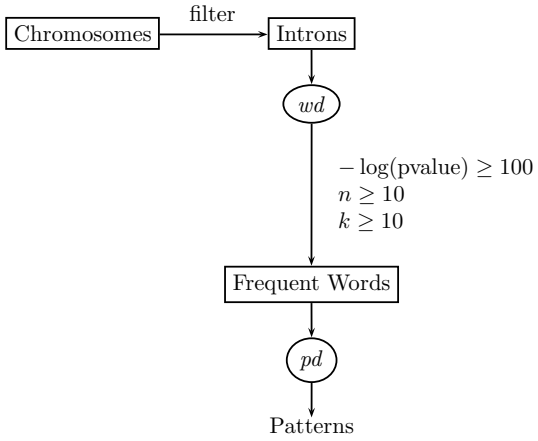


Figure 5: Drosophila melanogaster pattern discover process.

Pattern	Genome	Introns (p(genome))	Introns (p(introns))
GGACTGGGAC	64.83	916.01	105.17
GACTGGGACT	58.66	834.81	919.80
CTGGGACTGG	192.51	930.65	1052.61

Table 1: Patterns with length 10 and respective score under different distributions and sequences.

We then used the organism disjoint introns (sections of DNA that are spliced out after transcription but before the RNA is used) to build an input to the genetic algorithm. A word-discovery program was used for this purpose with $-\log(\text{pvalue}) \geq 100$, the number of occurrences being equal or greater than 10, the length at least 10 and requesting every word to have all the alphabet characters. For better understanding the process, see Figure 5.

The genetic algorithm was then ran with 64 individuals, with pattern length 27. The symbols probabilities were gathered from the disjoint introns sequence. The best pattern after 4096 generations was

ATTGTAAGTCTTTAAATATATTCGTGT

with a score of 7309.3774703, occurs 256 times and has a probability smaller than 10^{-9}. Curiously, this pattern is a sub-word of the consensus described in [7].

Having shown that the program is capable of finding known patterns we attempted to find new ones. The program was configured to run for 64 generations, with pattern length of 10 and 30.

Table 1 presents the patterns discovered with length 10. All floats were rounded to two significant digits. It is interesting to note that all patterns can potentially overlap all others. In Table 1 it is also shown the scores obtained by the patterns with different probability distributions and on different sequences, namely, on the whole genome with the distribution observed in the genome, on the introns with the distribution observed in the genome, and finally on the introns with the distribution observed in the introns.

The evaluation of the patterns in BLAST[1] returned no significant hits, as the lowest E-Value was only 0.11.

Some of the patterns found with length 30 were:

- ACG[AT][AT]A[AT][AT]A[AT][CT][AGT]A[AGT]TTTT AAA[AG]G[AG]TGGGCA
- [AC][AT]G[CT][AC]AA[AT]T[CG][GT][AG][CT]T[GT] AGG[AT][CG]A[AG]GC[CT]T[CT]AG[GT]
- TTAGCCACCCAACTTCTTTGGTAACTGGTG

[1] http://flybase.net/blast/

Sequence	Length (bp)
Saccharomyces cerevisiae (whole genome)	12156606
Anopheles gambiae (chromosome 2R)	61545105
Drosophila melanogaster (whole genome)	144141726

Table 2: Organisms used for evaluation.

The above patterns where evaluated with BLAST (at `http://flybase.net/blast/`) for the four most occurring words in the drosophila melanogaster genome that match the pattern. The BLAST returned significant (E-Value$< e^{-3}$) results on different organisms and chromosomes.

5 Empirical Evaluation

We study the sequential and parallel efficiency of BIORED by applying it to several datasets (sets of sequences). We also study the behavior of the GA in terms of convergence and execution time. The data sets used in the experiments are indicated in Table 2. They were obtained from the release 38 of the Ensembl project [13].

All experiments were ran in Dual core "AMD Opteron Processor 250" computers, with 4 gigabytes of RAM (but only 600 MB free), running the Linux operating system (kernel 2.6).

5.1 Performance and Algorithm Convergence

Figure 6 shows what happens to the runtime when we alter a single parameter, such as the population size or the pattern length. Theoretically the runtime is expected to double when the population size is doubled (see Algorithm 1 lines 15-26). However, the cut performed in line 19 makes the runtime vary.

The three organisms used (see Table 2) can be processed in about (largest to smallest) 27, 10 and 4 hours, running for one-thousand generations with a population size of 128 individuals and searching for patterns with length of 64.

When the pattern length is increased something apparently strange happens. Until a certain pattern length the runtime increases and then it decreases. This is, once again, related to the size of the search space. When the search space increases too much, the genetic algorithm has trouble in finding an admissible pattern. This makes the patterns being pruned by line 19 of Algorithm 1. A possible solution to this problem could be to initialize the population with statistically interesting words (naturally, found with another tool).

The runtimes when the population size is equal to 32 are very close to the ones with the population with 64 individuals. This is a consequence of the bit-parallelism technique implemented and for performing the runs in machines having 64-bit words.

The plots of the genetic algorithm convergence for the three organisms, indicated in Table 2, and for different population sizes and patterns lengths can be see in Figure 7.

From the results obtained we can conclude that when the pattern length increases, the population size must also be increased for the convergence to be smoother. This happens because it is more difficult to obtain an admissible large pattern. In the case of length=64 and population=32, the genetic algorithm cannot find a single pattern that occurs once in the sequences. This was expected, since the search space of a DNA pattern with length 64 is 2^{4*64}.

Furthermore, from Figure 7 is is clear that the smaller the pattern is, the faster the algorithm converges. This is rather expected since the search space is exponentially smaller.

The convergence values helped setting the parameter of the default number of generations. Because in 80% of the tests, at the five-hundred generation, it converged to 50% of the most fit individual of the at one-thousand generation.

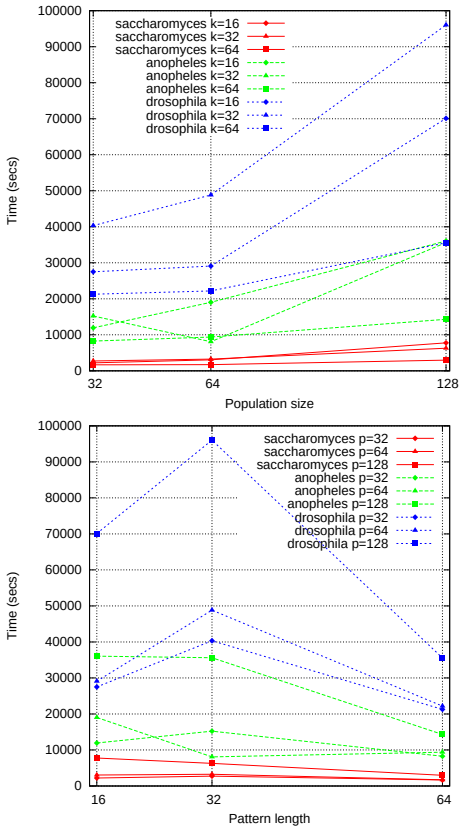


Figure 6: Run time variation with different populations and pattern lengths (in seconds).

5.2 Parallel speedup

The parallel speedup is plotted in Figure 8. The difference between the observed speedup and the optimum for Saccharomyces with 16 processors with parameters population=128 and pattern length=32 can be explained by the fact that different intervals have different amounts of work.

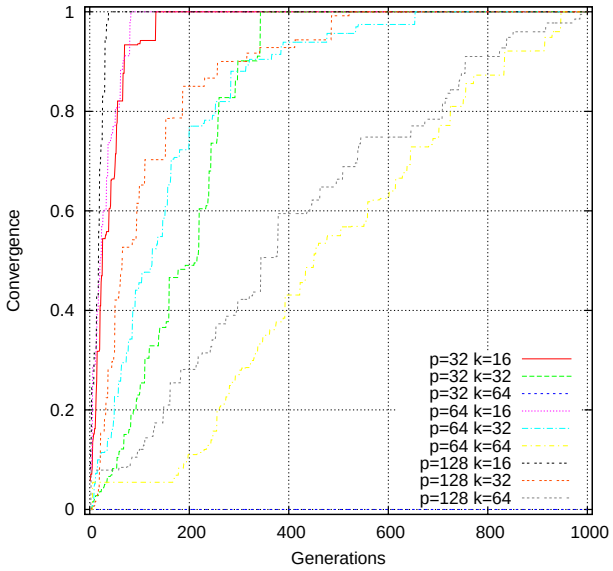
With the parallelization we were capable of running the slowest experiment in less than two hours. This is a significant improvement when compared this with the last section results, where the slowest required roughly 27 hours to complete.

These results lead us to believe that the human genome, with 3.2 billion base-pairs, can be mined with BIORED for patterns with length ≤ 64 in less than half a day, using a cluster with one hundred processors similar to those used for performing the experiments.

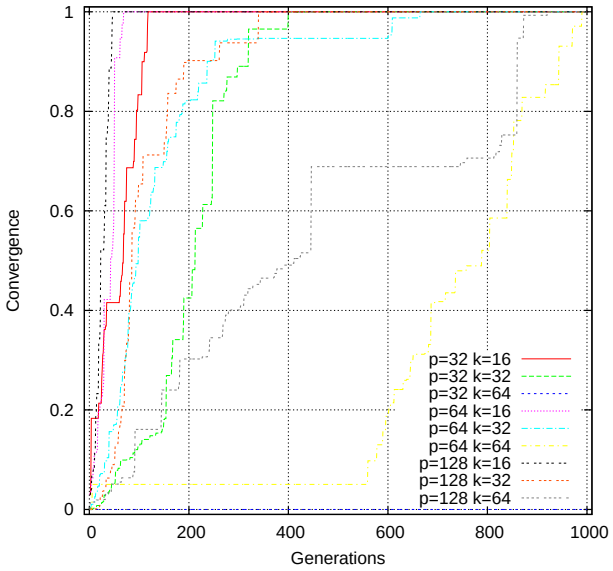
6 Related Work

Several pattern discovery tools and algorithms have been developed [21, 14, 22]. Some approaches are based on exhaustive search that guarantee that the best pattern (accordingly to some specifications) is found. An heuristic approach does not guarantee that the best pattern is found, instead they find a good "enough" pattern. The advantage of the heuristic approach is that it is often faster than the exhaustive search, but may not find the best solution (pattern).

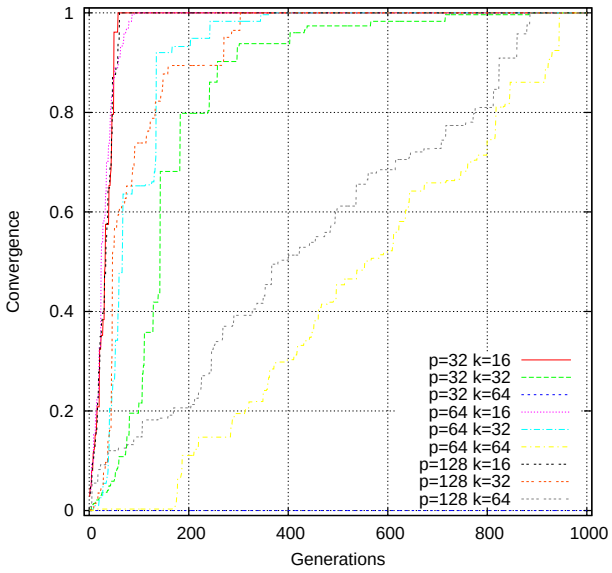
The Teiresias [21] is closely related to our proposal in terms of the pattern language. The Teresias is a program developed at the IBM Bioinformatics and Pattern Discovery group. It is based on well-organized exhaustive search based on combinations of shorter patterns. The Teiresias algorithm guarantees that all maximal [21] patterns are reported. The algorithm needs to be feed with the minimum number of literals that a pattern can contain, L. Another required parameter



a) Saccharomyces cerevisiae



b) Anopheles gambiae



c) Drosophila melanogaster

Figure 7: Genetic algorithm convergence while varying the population size (i) and the pattern length (k).

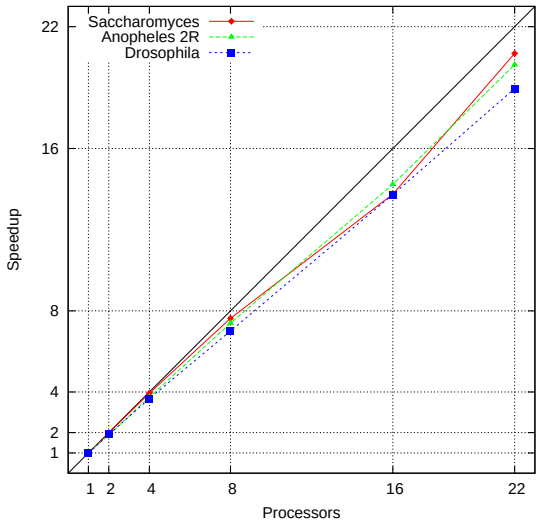


Figure 8: Parallel Speedup.

is W that indicates the maximum distance between any consecutive L literals. In general, if we use the same L parameter and increase the W parameter, the execution time of the scanning phase, which is the phase where the algorithm gathers seed patterns with the desired L and W characteristics, greatly increases. In the worst case the algorithm is $O(n^3 log\ n)$, but it is reported to work very well when the inputs are highly regular inputs and the parameters W and L are small.

The admissible patterns are similar to the ones we consider. The original Teiresias algorithm only supported one wild card equivalent to our ".". Newer versions support equivalency classes. In an equivalency class we specify the characters that are to be treated as equal in the actual pattern discovery process. These are similar to the classes of characters we support.

A critical problem with Teiresias is that it has a very high memory usage. In an attempt to evaluate the performance of Teiresias with classes of characters, we created the class file as follows:

```
AC, AG, AT, CG, CT, GT, ACG, AGT, AGT, CGT
```

and set the parameters $L = 8$, $W = 11$, convolution length equal to 2 and defined that a pattern should occur at least 48 times in the input. With these parameters, Teiresias crashed after 8 minutes with a memory consumption of several gigabytes. These parameters were chosen to verify if the algorithm could identify the previously discovered pattern, using the genetic algorithm, in the human gene of proinsulin.

Pratt [14] is a tool to discover patterns conserved in sets of unaligned protein sequences. The patterns that can be found are a subset of the patterns that can be described using Prosite notation [17]. In particular, variable length gaps are allowed. Pratt is very memory intensive, contrasting to the proposed genetic algorithm, which is pretty light in memory consumption. Pratt tries to find a pattern that occur in the most sequences as possible, while the program presented here considers the total number of occurrences in all sequences.

MEME (Multiple EM for Motif Elicitation) [2] uses a stochastic search to discover patterns. It does not require a pattern length parameter, which can be estimated by the algorithm itself. The algorithm is based on expectation maximization technique. Individual MEME patterns cannot contain gaps, and thus are equivalent to the patterns we consider. The overall complexity of MEME is quadratic in the size of the dataset and linear in the length of the pattern [2], while our proposal is linear in the size of the dataset and in the length of the pattern. A parallel version of the MEME, called ParaMeme [10], has been developed. It runs on specific parallel computer, while our parallel version runs on parallel computers and distributed computers. Another difference

between the two parallel algorithms resides in the parallelization strategy followed: ParaMeme follows functional parallelism strategy while we follow a data-parallelism strategy.

7 Concluding Remarks

We presented a new pattern discovery algorithm that discovers interesting patterns, in the form of a regular expression, using a genetic algorithm. The algorithm has a conservative memory usage of $O(ik|\Sigma|)$ and a worst-case time complexity of $O(nikg)$, where Σ is the alphabet used, i is the number of individuals of the population, k is the length of the pattern, n is the size of the input, and g is the number of generations. However, the algorithm is in average much faster, achieving a complexity of $O(gni/w)$, where w is the number of bits in a machine word. The average complexity is directly linked to the average case of the naive string matching. Experiments showed the usefulness of the algorithm, by demonstrating that it is capable of finding previously known patterns. Moreover, the parallel version is capable of achieving almost linear speedups on a distributed memory computer.

Some questions still remain. First, the initial population is currently generated randomly, covering the entire range of possible patterns. We plan to reduce the initial diversity by initializing the population with initial patterns where optimal solutions are likely to be found. Second, we plan to extend BIORED to perform classification. Finally, we plan to make BIORED available on the WWW. For the moment it is available upon request by contacting the corresponding author.

Acknowledgments

We thank Jorge Vieira for providing the annotated Drosophila dataset. The work presented in this paper has been supported by funds granted to *LIACC* through the *Programa de Financiamento Plurianual, Fundação para a Ciência e Tecnologia* and *Programa POSI*. Pedro Pereira is funded by FCT grant SFRH/BD/30628/2006.

References

[1] NCBI Basic Local Alignment Search Tool. `http://ncbi.nih.gov/blast/`.

[2] Timothy L. Bailey and Charles Elkan. Fitting a mixture model by expectation maximization to discover motifs in biopolymers. In *Proceedings of the Second International Conference on Intelligent Systems for Molecular Biology*, pages 28–36, Menlo Park, California, 1994. AAAI Press.

[3] Dennis A. Benson, Ilene Karsch-Mizrachi, David J. Lipman, James Ostell, and David L. Wheeler. Genbank. *Nucleic Acids Research*, 33:235–242, 2005.

[4] Helen M. Berman, John Westbrook, Zukang Feng, Gary Gilliland, T. N. Bhat, Helge Weissig, Ilya N. Shindyalov, and Philip E. Bourne. The protein data bank. *Nucleic Acids Research*, (28):235–242, 2000.

[5] Greg Burns, Raja Daoud, and James Vaigl. LAM: An Open Cluster Environment for MPI. In *Proceedings of Supercomputing Symposium*, pages 379–386, 1994.

[6] Richard Cole and Ramesh Hariharan. Verifying candidate matches in sparse and wildcard matching. In *STOC '02: Proceedings of the thiry-fourth annual ACM symposium on Theory of computing*, pages 592–601, New York, NY, USA, 2002. ACM Press.

[7] Javier Costas, Cristina P. Vieira, Fernando Casares, and Jorge Vieira. Genomic characterization of a repetitive motif strongly associated with developmental genes in drosophila. *BMC Genomics*, 2003.

[8] Willian Feller. *An Introduction to Probability Theory and Its Applications*. John Wiley & Sons, third edition, 1968.

[9] Nuno A. Fonseca, Rui Camacho, and Alexandre Magalhães. Relevance of short sequences of amino acid residues at the N- and C-termini of helical segments in proteins. *Submited to PROTEINS: Structure, Function and Bioinformatics*, 2006.

[10] W. Grundy, T. Bailey, and C. Elkan. Parameme: A parallel implementation and a web interface for a dna and protein motif discovery tool. *Computer Applications in the Biosciences*, 12(4)(303-310), 1996.

[11] Dan Gusfield. *Algorithms on strings, trees, and sequences: computer science and computational biology*. Cambridge University Press, New York, NY, USA, 1997.

[12] H.M.Berman, J.Westbrook, Z.Feng, G.Gilliland, T.N.Bhat, H.Weissig, I.N.Shindyalov, and P.E.Bourne. The protein data bank. *Nucleic Acids Research*, pages 235–242, 2000.

[13] T. Hubbard, D. Andrews, M. Caccamo, G. Cameron, Y. Chen, M. Clamp, L. Clarke, G. Coates, T. Cox, F. Cunningham, V. Curwen, T. Cutts, T. Down, R. Durbin, X. M. Fernandez-Suarez, J. Gilbert, M. Hammond, J. Herrero, H. Hotz, K. Howe, V. Iyer, K. Jekosch, A. Kahari, A. Kasprzyk, D. Keefe, S. Keenan, F. Kokocinsci, D. London, I. Longden, G. McVicker, C. Melsopp, P. Meidl, S. Potter, G. Proctor, M. Rae, D. Rios, M. Schuster, S. Searle, J. Severin, G. Slater, D. Smedley, J. Smith, W. Spooner, A. Stabenau, J. Stalker, R. Storey, S. Trevanion, A. Ureta-Vidal, J. Vogel, S. White, C. Woodwark, and E. Birney. Ensembl 2005. *Nucleic Acids Research*, 33(1), January 2005.

[14] Inge Jonassen, John F. Collins, and Desmond Higgins. Finding flexible patterns in unaligned protein sequences. *Protein Science*, 4(8):1587–1595, 1995.

[15] Amy Lew, William J. Rutter, and Giulia C. Kennedy. Unusual dna structure of the diabetes susceptibility locus iddm2 and its effect on transcription by the insulin promoter factor pur-1/maz. *Proceedings of the National Academy of Sciences of the United States of America*, 97(23):12508–12512, November 2000.

[16] Zbigniew Michalewicz. *Genetic Algorithms + Data Structures = Evolution Programs (Third, Revised and Extended Edition)*. Springer-Verlag, New York, NY, USA, 1999.

[17] Hulo N., Bairoch A., Bulliard V., Cerutti L., De Castro E., Langendijk-Genevaux P.S., Pagni M., and Sigrist C.J.A. The prosite database. *Nucleic Acids Res.*, 34(D227-D230), 2006.

[18] Gonzalo Navarro. Pattern matching. *Journal of Applied Statistics*, 31(8):925–949, 2004. Special issue on Pattern Discovery.

[19] Gonzalo Navarro and Mathieu Raffinot. *Flexible Pattern Matching in Strings*. Cambridge University Press, New York, NY, USA, 2002.

[20] R Development Core Team. *R: A language and environment for statistical computing*. R Foundation for Statistical Computing, Vienna, Austria, 2005. ISBN 3-900051-07-0.

[21] Isidore Rigoutsos and Aris Floratos. Combinatorial pattern discovery in biological sequences: The teiresias algorithm. *Bioinformatics*, 14(1):55–67, 1998.

[22] S. Sinha and M. Tompa. An exact method for finding shor motifs in sequences, with application to the ribosome binding site problem. *Proceedings of the 7th International Converence on Intelligent Systems for Molecular Biology (ISMB)*, pages 262–271, 1999.

[23] S. Sinha and M. Tompa. A statistical method for finding transcription factor binding sites. *Proceedings of the National Academy of Sciences of the United States of America*, 95(6):2738–2743, 2000.

[24] J. van Helden, M. del Olmo, and J. Perez-Ortin. Statistical analisys of yeast genome downstream sequences reveals putative polyadenylation signals. *Nucleic Acids Research*, 28(4):1000–1010, 2000.

[25] G. Wang and Jr. R. L. Dunbrack. Pisces: a protein sequence culling server. *Bioinformatics*, 19:1589–1591, 2003.