An algorithm to find all identical internal sequence repeats

Nirjhar Banerjee¹, N. Chidambarathanu¹, Daliah Michael¹, N. Balakrishnan² and K. Sekar¹,²,*

¹Bioinformatics Centre (Centre of Excellence in Structural Biology and Bio-computing) and
²Supercomputer Education and Research Centre, Indian Institute of Science, Bangalore 560 012, India

Proteins containing amino acid repeats are considered to be of great importance in evolutionary studies. The principal mechanism of formation of amino acid repeats is by the duplication or recombination of genes. Thus, repeats are found in both nucleotide and protein sequences. In proteins, repeats are involved in protein–protein interactions as well as in binding to other ligands such as DNA and RNA. The study of internal sequence repeats would be helpful to scientists in various fields, including structural biology, enzymology, phylogenetics, genomics and proteomics. Hence an algorithm (Finding All Internal Repeats, FAIR) has been designed utilizing the concepts of dynamic programming to identify the repeats. The proposed algorithm is a faster and more efficient method to detect internal sequence repeats in both protein and nucleotide sequences, than those found in the literature. The algorithm has been implemented in C++ and a web-based computing engine, IdentSeek, has been developed to make FAIR accessible to the scientific community. IdentSeek produces a clear, detailed result (including the location of the repeat in the sequence and its length), which can be accessed through the world wide web at the URL http://bioserver1.physics.iisc.ernet.in/ident/

Keywords: Dynamic programing, evolutionary studies, internal sequence repeats, structure–function relationship.

INTERNAL repeats occur in approximately 14% of all known proteins available in the literature. In fact, eukaryotic proteins are three times more likely to contain internal sequence repeats than prokaryotic proteins, because eukaryotic repeats have unique functions. In addition, eukaryotic genomes have undergone a higher degree of gene duplication. Since gene duplication and recombination events are thought to be responsible for the existence of sequence repeats, they are found in both genes and non-coding genomic regions. In protein sequences, repeats can range from a single residue to complete domains consisting of hundreds of amino acid residues. Repeats that are separated by intermediate sequences of constant length occurring in clusters are referred to as short regularly spaced repeats (SRSRs). Since most of these are identical repeats, they can be effectively applied to protein secondary-structure prediction.

Earlier studies indicated that the topology of transmembrane proteins evolved by internal gene duplication and a large number of them have been detected to contain internal sequence repeats. As stated earlier, the secondary structures of such internal repeats in proteins are regular. A classic example of internal repeats is the zinc-finger domain, which is a commonly repeated motif in most DNA-binding proteins. Internal sequence repeats also have important implications in evolution and functions of proteins. In his book, Of Flies, Mice and Men, Francois Jacob has put forth that evolution has arisen from nature’s tinkering, implying that nature makes new genes with new functions from existing genes. Kachroo et al. suggested that the formation of internal sequence repeats could be the major cause of evolutionary genome variability. Furthermore, in evolutionary studies, the distance of the constituent repeats can use information from internal repeats to estimate the age of a genome or an individual protein. In addition, insights into evolution and phylogenetic relationships can be derived from the distribution of repeats. For instance, Koonin et al. concluded that the distribution of repeats in archaea indicates that they have an intermediate relationship between prokaryotes and eukaryotes. Earlier studies indicate that repeats play a major role in protein and nucleotide stability, thus being responsible for protein function. Internal repeats not only ensure the proper functioning of proteins, but sometimes cause malfunction and hence are important in the study of diseases, such as Alzheimer’s disease, Creutzfeldt–Jacob disease and Gerstmann-Straussler–Scheinker disease.

In the case of DNA sequences, short repeats indicate the position of deletion and precise removal of transposable elements. Longer repeats are responsible for class switching in immunoglobulins. Further, WD40 repeats in CARD4 act as regulators for the activation of caspases in apoptosis pathways. The activity of enzymes can also be controlled to some extent by internal sequence repeats. For example, decay accelerating activity is reduced if repeats between sites 1 and 2 are deleted in C5 conver-
sequences\textsuperscript{18}. Further, in the case of cGMP-binding phosphodiesterases there are two repeats, each of which is a catalytic site and an invariant Aspartate residue is essential for interaction\textsuperscript{19}. Furthermore, in case of nucleotide repeats, telomeric repeats are essential to ensure correct function of telomeres and for maintaining the integrity and stability of the chromosome\textsuperscript{20}. Thus, the significance of internal sequence repeats is clear.

**Existing algorithms**

Many algorithms have been developed to search for sequence repeats in protein sequences. These methods use different methodologies and protocols to detect internal sequence repeats. According to Söding\textit{ et al.}\textsuperscript{21}, there are three general classes among the existing methods to detect repeats in protein sequences. SMART\textsuperscript{22} and REP\textsuperscript{23} belong to the first class, which construct repeats by utilizing their own database of profile-hidden Markov models or sequence profiles generated from known repeat families and comparing each profile with the query sequence. REPPER\textsuperscript{24} belongs to the second class which is specialized for the detection of periodic patterns in proteins. This class does not allow gaps within or between repeats and is applicable mostly to the large class of fibrous proteins. Although REPwin in REPPER is similar to other tools, it detects short repeats by aligning the sequence against itself and deploying a sliding window protocol. However, these short repeats are detected only if they occur consecutively in sufficient number in the sequence to be detected by Fourier transformation. The third class consists of REPRO\textsuperscript{25}, RADAR\textsuperscript{26} and TRUST\textsuperscript{27}. These do not rely on any a priori knowledge about repeat families, but look for internal similarities by aligning the sequence with itself. RADAR results in short composition-biased repeats together with gapped approximate repeats. TRUST generates only those repeats that are statistically significant. The program REPuter\textsuperscript{28} finds repeats only in nucleotide sequences by implementing suffix trees.

Most of these methods concentrate on homology and produce repeats more than 15 residues that frequently contain gaps. Thus, small repeats may be missed by some of these programs. While some programs allow mismatches to account for the mutations in sequences, and are a good design choice for the detection of domain repeats across different proteins, they have the disadvantage of sometimes showing completely unrelated sequences as repeats. Additionally, the main limitation of these algorithms is the limit imposed on the maximum number of amino acid residues that can be given as input. RADAR web-interface permits a maximum of 1000 amino acid residues, Internal Repeats Finder\textsuperscript{29} permits 2000 amino acid residues and REPRO allows a maximum of 6000 amino acid residues. However, REPPER has no limits on the number of amino acid residues in a given sequence, but permits only one protein sequence at a time, completely ignoring any later sequences provided in the input. Further, in case of RADAR, TRUST and REPRO, when multiple protein sequences are entered, the algorithm treats the entire set of sequences as a single sequence.

To address these issues, an algorithm FAIR (Finding All Internal Repeats) has been proposed to search for all identical internal sequence repeats. In contrast to the algorithms described earlier, the proposed algorithm utilizes the concepts of dynamic programing rather than simple sequence alignment to find the internal identical repeats in both protein and nucleotide sequences. Unlike BLAST, which is a local alignment search tool and whose output must be heavily parsed to find internal repeats in sequences, the proposed algorithm finds and individually lists all the internal repeats present in the sequence. Further, the algorithm runs well for finding precise results from extremely large sequences, as described in later sections.

**The proposed algorithm**

The algorithm to find identical internal sequence repeats in protein and nucleotide sequences is described below. The time taken for execution and the memory utilized while running the program have been optimized and are explained in the following sections. As the number of strings repeated and the number of times one particular string is repeated are not known, size allocation of arrays is not preferred, and thus the simple dynamic programing concepts are utilized in this algorithm. The algorithm consists of three parts and is explained in detail in the subsequent sections.

**Finding the repeats using two vectors, ‘current’ and ‘previous’**

The uploaded sequence or string of length \( y \) is named \( a_1 \) and an identical string \( a_2 \) is created for calculation purposes. Two vectors named ‘previous’ and ‘current’ of \( y + 1 \) elements each are created and initialized to zero. The \( y + 1 \)th element in the vector is required in order to check for the end of the string. Although there is no alignment involved, the procedure to find identical internal repeats in a given sequence can be visualized using a graph, where the string \( a_1 \) is along the \( y \)-axis and \( a_2 \) is along the \( x \)-axis (Figure 1). For each character of \( a_1 \), the algorithm checks all the characters of \( a_2 \). In effect, a match between the character denoting the row (in string \( a_2 \)) and the one denoting the column (in string \( a_1 \)) is searched. Since the matrix is symmetrical, it is sufficient to check only the upper half of the main diagonal. In other words, if \( i \) denotes the position of the pointer in \( a_1 \), the algorithm checks from the \( i + 1 \)th element of \( a_2 \) till the end of the corresponding row and whenever a match is found, the following operation is performed:
if (a1[i] equals a2[j]){  
    set current[j] to previous[j - 1] + 1; }

The above step assigns the current length of the repeat to the jth element of vector ‘current’. While performing the next iteration, the vector ‘previous’ is assigned the value of vector ‘current’ and the above step is repeated. For example, if the string ‘ABC’ (position 6–8) is repeated twice, with the second repeat from positions 11 to 13 in the vector, the changes in the required elements of both vectors will be as shown below:

1. Initially, current = 0; previous = 0.
2. After the first match A: current[10] = 1; previous=0.
3. Then, previous is assigned the value of current:  
   previous[10] = 1; current[10] = 1; rest = 0.
4. After the character B is found: current[11] = 2; no other changes.
5. Similarly, after the character C is found: current[12] = 3.

Thus, the ‘current’ vector stores both the position and length of the repeat. In this case, we find that 3 is the length of the repeat and the 13th position is the ‘end point’. The algorithm first checks whether the last length of the repeat (if there is one) that is stored in the ‘previous’ vector, is greater than the minimum length (min1) of the repeat defined by the user. If this condition is satisfied, the position and length of the repeat are pushed into another vector, ‘vsparse’. Subsequently, the vector ‘vsparse’ stores the user-defined data-type structure called ‘SparseEntry’ which in turn stores three variables, corresponding to the position, end (row_value and column_value) and length (length_value) of the repeat. Thus, the algorithm performs the following operation whenever a match is not found and if the end of a row is not reached:

if(previous[j - 1] is greater than or equal to min1){  
    set sparse_entry_column_value to j - 1;  
    set sparse_entry_row_value to i - 1;  
    set sparse_entry_length_value to previous[j - 1];  
    push sparse_entry into vector vsparse; }

Once the end of the row is reached and the condition previous[j-1]>=min1 is satisfied, the algorithm stores the values in the vector as mentioned earlier. Thus, the values of the ‘end-points’ and the length of the repeats have been computed and stored at the end of the first part of the algorithm. When more than one sequence is given as input, the repeat is stored and the vector current is reset to 0. Further, whenever a new sequence is encountered, the vector ‘previous’ is set to the value of ‘current’; hence it is also reset to zero.

Storing subsequences and repeat positions

When the length of the repeat (stored in vsparse.value) is greater than the minimum length, the vectors ‘startd’ and ‘endd’ are created based on the contents of vsparse. This can be explained using the string ‘ABC’, which is repeated between positions 6 and 8 as well as 11 and 13 (Figure 1). Thus, the vector ‘startd’ contains the positions of the starting point of the ‘first’ and the ‘second’ sequences. Similarly, the vector ‘endd’ contains the positions of the end-points of the two sequences. If there are more than two occurrences of a particular string, the ‘startd’ and ‘endd’ values of the ‘second sequence’ for further occurrences need to be stored, but not those of the ‘first sequence’. To accomplish this, the algorithm uses a Boolean variable append_first, which is false unless it encounters a new repeat.

if (((endd of first sequence is not equal to lrow) OR (seqlen is not equal to vsparse[i].value)) {  
    set append_first to true;  
    set seqlen to vsparse[i].value;  
    set lrow to endd[firstseq]; }

if(append_first is true){  
    set sub_sequence.start_point to startd[firstseq];  
    set sub_sequence.end_point to endd[firstseq];  
    push sub_sequence into vsubseq; }

Figure 1. Visualization of the procedure to detect internal repeats (ABC) from a sequence. a1 and a2 are the two vectors containing the substring ABC, which is repeated between positions 6 and 8 as well as 11 and 13. ‘Startd’ and ‘endd’ contain the starting and ending positions of the substring respectively.
set sub_sequence.start_point to startd[secondseq];
set sub_sequence.end_point to endd[secondseq];
push sub_sequence into vsubseq;

Whenever either the end-points of two consecutive repeats or their lengths are different, append_first is set to true. A new vector 'vsubseq' stores a data structure called 'SubSequence' containing three elements: sequence (to store the repeat) and start_point and end_point (for the initial and terminal points of the repeat respectively). Thus, when a new repeat is encountered, the positions of both the 'first' and 'second' sequences are pushed into the vector 'vsubseq'. In other cases, only the positions of the 'second' sequence are pushed.

Sorting the vector and removing identical entries

The entries of the vector 'vsubseq' are entered as and when the repeats are encountered. Thus, the results must be sorted to make the output non-redundant. This is performed using the built-in-function STL sort (NlogN), implemented in C++. Further, in order to remove redundant entries from the sorted output every element is checked with the previous one and if two subsequent entries are found to be the same, the second one is removed. The non-redundant entry is pushed into a vector called 'unique_entry', which displayed as the output. Interestingly, none of the generated sets of repeats is a subset of another. To illustrate this aspect, suppose the sequence 'ABCD' is repeated three times, it would also mean that the sequence 'ABCD' is repeated three times. However, this second repeat ('ABCD') will not be shown unless there is an independent repeat of 'ABCD'. Thus, the algorithm is designed to show only non-redundant repeats. However, in the case of low-complexity repeats the algorithm generates all subsets as repeats, irrespective of whether they occur independently in other parts of the sequence.

While FAIR follows O(N^2) time complexity in the generalized case, where N is the number of amino acids present in the sequence, TRUST follows O(N^2 + NA^2 + TLN) complexity, which takes less than a minute to find repeats in 2000 residues27. While the algorithm used in RADAR takes not more than 180 s to find repeats in 5179 amino acid residues (e.g. MUC2_HUMAN) and 72 s to identify repeats in a sequence of length 2265 residues (e.g. FINC_BOVIN)26, the proposed algorithm takes less than a second to find repeats from these sequences. Further, FAIR takes only 32 s to find repeats in the largest protein sequence 'Titin' from Homo sapiens (gi|17066105|emb|CAD12456.1), which contains 34,350 amino acid residues. Thus, the time taken by the proposed algorithm is less compared to those available in the literature. However, keeping in mind the time complexity, the algorithm (and web-based computing engine) performs best with a minimum repeat length of three amino acids.

Implementation: IdentSeek, a web-based computing engine

In order to make the proposed algorithm freely available to the scientific community, a web-based computing engine, IdentSeek, has been developed. The computing engine is written in Perl/CGI and can handle extremely large sequences (FASTA format). It has been successfully tested with the nucleotide sequence of the parallel β-helix repeat containing protein from Chlorobium chlorochromatii CaD3 (gi|78188592; approximately 37,000 amino acids; 110,000 nucleotides). In addition, it is possible to distinguish between different protein sequences when multiple such sequences are uploaded and to detect all identical repeats irrespective of restrictions on their composition or alignment. Further, it is flexible and allows the user to set the required length of the repeat. Finally, it produces a complete and comprehensive output for each of the sequences.

Case studies

To test the efficiency of the proposed algorithm, an invasion from Yersinia pestis Nepal516 containing 4270 amino acid residues was taken in FASTA format. The minimum number of amino acid residues in a given repeat was set to greater than or equal to 60. For clarity, only a portion of the input protein sequence is shown in Figure 2. It is evident from the output that there are two distinct

Figure 2. Output of FAIR for protein sequence using an invasion from Yersinia pestis.

<table>
<thead>
<tr>
<th>Peptide</th>
<th>Start</th>
<th>Size</th>
</tr>
</thead>
<tbody>
<tr>
<td>(gi</td>
<td>108813752)ref[YP_649519.1] invasin [Yersinia pestis Nepal516]</td>
<td>1002 to 1065</td>
</tr>
<tr>
<td>(gi</td>
<td>108813752)ref[YP_649519.1] invasin [Yersinia pestis Nepal516]</td>
<td>2452 to 2515</td>
</tr>
</tbody>
</table>

Number of residues in the repeat = 64

Number of residues in the repeat = 248

Figure 3. Output of TRUST server for part of the hypothetical protein from Y. pestis.

<table>
<thead>
<tr>
<th>Peptide</th>
<th>Start</th>
<th>Size</th>
</tr>
</thead>
<tbody>
<tr>
<td>(gi</td>
<td>17066105)emb</td>
<td>CAD12456.1</td>
</tr>
<tr>
<td>(gi</td>
<td>17066105)emb</td>
<td>CAD12456.1</td>
</tr>
</tbody>
</table>
identical repeats containing 64 and 248 amino acid residues respectively. The same sequence was submitted to the existing algorithms. The servers RADAR, TRUST and Internal repeat finder did not accept the input sequence due to the limitations in the number of amino acid residues. Further, REPPER server did not produce any significant results. In addition, there was a problem in obtaining results from REP and REPRO due to internal server error. The algorithm RepPuter was not used for this sequence, since it has been developed for nucleotide sequences.

Since the algorithms RADAR, TRUST and Internal repeat finder have limitations in the number of amino acid residues given in the input, the protein fragment from residue numbers 251 to 575 (325 amino acid residues) of the hypothetical protein YPN_1947 from \textit{Y. pestis} Nepal516 (\texttt{gi|108812109|ref|YP_647876.1|}) was given as input. TRUST yielded the first 159 residues as a repeat, occurring in succession and hence allowing for mismatches (Figure 3). REPwin in REPPER resulted in composition-based repeats (Figure 4). RADAR produced gapped repeats, also allowing a certain degree of mismatch in the repeats. It also generated sequences which are subsets of a larger repeat, hence resulting in redundancy (Figure 5).

In contrast, IdentSeek, the computing engine implementing the algorithm FAIR gave 11 non-redundant identical sequence repeats (Figure 6). In contrast to FAIR, which takes one or more sequences simultaneously, the commercially available Sequence Analysis Software Package of the University of Wisconsin Genetics Computer Group \cite{UWGCG} (UWGCG) ‘repeat’ function can be run for only one sequence at a time. Each time the function is invoked, the minimum window size, stringency and other relevant parameters must be set. Thus, utilization of this program requires some initial time for learning. In addition, the FASTA format files must be converted to GCG format using the function ‘fromfasta’. Additional parsing of the output, which allows mismatches depending on the stringency criteria (Figure 7) is necessary to obtain the location of the repeats. In this particular case, it is clear that the default criteria are insufficient to find identical repeats as the protein fragment mentioned above yields 100 repeats of lengths starting from two residues, not all of which are identical. On the other hand, although BioSuite \cite{BioSuite} has a repeat analysis function in the pattern identification and matching sub-menu, this function does not appear to work for a large number of proteins or for protein sequences containing a large number of amino acids.
Figure 6. Output display of the proposed algorithm, FAIR for the same 325 residue protein fragment from Y. pestis.

Figure 7. Output of UWGCC for the 325 residue protein fragment from Y. pestis. Only part of the output is shown for clarity.

acid residues (e.g. Titin). Furthermore, although it gives the location of the repeat like FAIR, with the default parameters, it found only three repeats in the 325 residue fragment (Figure 8).

Furthermore, FAIR (and thus, IdentSeek) can be used to perform analysis on both protein and nucleotide sequences, whereas the other tools can only work for either
The proposed algorithm identifies all the identical sequence repeats in both protein and nucleotide sequences and produces a comprehensive output by displaying the number of repeats with their start and end positions. Due to dynamic allocation of memory, the memory required for the program is exactly equal to the memory required to store the repeats and thus, the algorithm is more efficient than FAIR. However, the REPuter server did not show the remaining two repeats (marked as ** in Figure 10) containing 542 and 142 nucleotides as produced by FAIR, as it displayed them in FASTA format. As the input sequence contains 68,532 nucleotides, only a part of the input sequence is shown in Figure 9 and the minimum number of nucleotides per repeat was set as greater than or equal to 100. The program produced eight independent identical repeats containing between 116 and 542 nucleotides. REPuter, the only freely available program for finding repeats in nucleotide sequences (other than FAIR), displayed six unique repeats (marked as * in Figure 10), similar to that of the repeats produced by FAIR. Since the proposed algorithm is designed to show only identical repeats, the program FAIR produces eight unique identical repeats.

### Conclusion

The proposed algorithm identifies all the identical sequence repeats in both protein and nucleotide sequences and produces a comprehensive output by displaying the number of repeats with their start and end positions. Due to dynamic allocation of memory, the memory required for the program is exactly equal to the memory required to store the repeats and thus, the algorithm is more efficient than FAIR.
cient in dealing with sequences having a large number of amino acid residues. This algorithm is unique in its utilization of dynamic programming rather than local alignment and in its speed compared to other algorithms. Furthermore, the algorithm has been implemented as a web-based computing engine, IdentiSeek, designed for giving the user a precise, clear and usable output. This would help in the analysis of protein sequences, especially to facilitate the study of evolutionary history of proteins and diseases related to genes and proteins. The computing engine is available on the world wide web at http://bioserver1.iisc.ernet.in/identek/.


ACKNOWLEDGEMENTS. We acknowledge the use of the Bioinformatics Centre, the Interactive Graphics Based Molecular Modeling facility and the Supercomputer Education and Research Centre at IISc. We thank the Department of Information Technology for funding this project. Part of this work is supported by the Department of Biotechnology sponsored Institute wide computational biology programme.

Received 16 November 2007; revised accepted 11 June 2008