Mathematics Today Vol.34(A) (April 2018 – Special Issue) 163-172
ISSN 0976-3228, e-ISSN 2455-9601

Iterative Matching of DNA Sequence

Nita H. Shah[1], Thakkar Vishnuprasad D.[2]

Department of Mathematics, Gujarat University, Ahmedabad 380009, Gujarat, India
Email: [1]nitashah@gmail.com [2]vd.thakkar@gmail.com

Abstract

The canonical structure of DNA has four bases: Thymine (T), Adenine (A), Cytosine (C) and Guanine (G). In research of DNA structures, DNA sequences are compared in a special way. Both the sequences may have a common sub-sequence and each of them may have additional base elements. Two sequences are considered as matching if common sub-sequence is more than threshold fraction like 80% length of the DNAs is matching. Frequency distribution and basic string matching does not work here. Basic process of finding matching largest sub-string is employed recursively. First largest matching sub-string in both the sequences is identified. This gives one matching sub-sequence and left and right un-matched subsequences from both the sequences. The same process is repeated recursively for the corresponding subsequences of both the sides (left unmatched subsequence of both the sequences and right unmatched of both the subsequences) are processed. This will help in identifying genetic diseases like Cancer, Thalassemia, etc. This approach can be used in search engines with dilution in ordering so that strings may match even if they do not maintain order (Like 'DNA Sequence Matching' and 'Matching .of DNA Sequences' match).

Keywords: DNA Sequence; String Processing; Recursion
AMS Subject Classification: 92D20

1. Introduction:

The canonical structure of DNA has four bases: Thymine (T), Adenine (A), Cytosine (C) and Guanine (G). DNA sequencing is the process of determining the precise order of nucleotides within a DNA molecule [1]. The advent of rapid DNA sequencing methods has greatly accelerated biological and medical research and discovery.

Knowledge of DNA sequences has become indispensable for basic biological research, and in numerous applied fields such as medical diagnosis, biotechnology, forensic biology, virology and biological systematics. The rapid speed of sequencing attained with modern DNA sequencing technology has been instrumental in the sequencing of complete DNA sequences, or genomes of numerous types and species of life, including the human genome and other complete DNA sequences of many animal, plant, and microbial species.

The first DNA sequences were obtained in the early 1970s by academic researchers using laborious methods based on two-dimensional chromatography. Following the development of fluorescence-based sequencing methods with a DNA sequence, DNA sequencing has become easier and orders of magnitude faster [1].

In research related to genetic behavior, sequences are matched. In this type of work, one important way of matching sequences is to match discarding non-matching extra elements.

One of the authors was introduced to the problem of DNA sequence matching. The objective was to find the matching segments in both the sequences discarding unmatched/ extra elements and for the specific problem, there was no penalty for gaps and mismatches. Measure of matching was number of matching elements in both the sequences maintaining order.

The following example illustrates the process.

We have generated one base DNA Sequence using Random DNA Sequence Generator available on University of California, Riverside website [2]. In the sequence, we have added elements randomly and generated two sequences shown in equations (1) and (2).

CTTCGAGAGATGAAGTATGTCCGGTCGGCCAGTA (1)

TATTCGAGATGAGAAATGTCTCCGGACGGCGTCAA (2)

2. Matching of sequence

We already know that the base sequence is common in both the sequences. Extra elements added in the each of the sequence are mismatches. Matched and unmatched elements are listed in Table 1. For easy identification, we have listed matched and unmatched sub-strings in lowercase and uppercase respectively.

C	ttcgaga	GA	tga		a	GT	atgtc		cgg	T	cggc		ca	GT	A
TA	ttcgaga		tga	G	a	A	atgtc	TC	cgg	A	cggc	GT	ca		A

Table 1: Result of Sequence Matching

When a sequence is matched with set of sequences, different sequences match to different extents. To order the relationship with the sequences, matching result has to be translated to a score. This is achieved by forming a super-sequence which is extension of both the sequences. When super-sequence is compared with our sequence, extra bases in super-sequence are mapped to – in the original sequence. This is illustrated in Table 2. Super-sequence is shown in first row. Other data sequences are shown in second and third row respectively. Super-sequence is used only to identify gaps and aligning both the sequences for matching by individual position.

T	A	ttcgaga	GA	tga	G	a	A	T	atgtc	TC	cgg	A	cggc	GT	Ca	GT	a
C	^	ttcgaga	GA	tga	^	a	G	T	atgtc	^^	cgg	T	cggc	^^	Ca	GT	a
T	A	ttcgaga	^^	tga	G	a	A	^	atgtc	TC	cgg	A	cggc	GT	Ca	^^	a

Table 2: Result of Sequence Matching with gaps, matches and mismatches

The final score is derived by assigning positive value for matched position and negative or neutral values for mismatches and gaps. Value for matching position 1, mismatch (-1 or 0) and gap with value -1 can be taken as described in [3] and [4]. Possibly mismatch of amicable bases (A-C and G-T) can be assigned intermediate value like 0.5.

We shall be building the DNA sequence matching algorithm by analysis of the solution.

As first step, we find the largest matching sub-string (we are not using term sub-sequence to avoid misinterpretation due to definition of sub-sequence in Mathematics). The matching sub-string is (*TTCGAGA*) in this case. This sub-sequence starts from position 2 in first sequence (1) and position 3 in second sequence (2).

This splits both the sequences in 3 parts namely unmatched string on left, matched string and unmatched part on the right. We apply the same method for processing left unmatched part of both the sequences as well as right unmatched part of both the sequences. If one of the sequences to be compared (not both) is empty (of zero length), they are unmatched. If both are empty, then nothing is to be done.

In this example, left unmatched part of first and second sequence is un-matched. For the right part, we find largest matching sub-string common in them. In this case (*ATGTC)* is matching.

Like in the first step, this step leaves unmatched sub-strings in both the sequences. Point to be noted here is that now sequences to be matched are right side unmatched sub-strings of both the sequences. In this step, we get matched string (*ATGTC)* and unmatched sub-strings on left side of both the sequences and unmatched sub-strings on right side of both the sequences. We repeat the process for corresponding unmatched sub-strings.

It is possible that two nonempty sequences (for example CGC and ATTA) may not have anything in common. In this case, sequences are totally different and no further process is done.

Possible outcomes of finding largest matching sub-string are

- One of sequences is empty - no further process is done. If both the strings are empty, nothing to be done. If one of them is non-empty, put the same in super-sequence. The parent sequence has this as gap.
- Sequences have nothing in common - no further process is done. For both the strings, search of sub-string of maximum length (smaller of both the string lengths) to single length character are searched in the second string and every time sub-string is not found in second string. Put larger of the two in the super-sequence. Parent sequence of larger sequence has gaps.
- Common sub-string is found. Result has this sub-string as matched sub-string; further repeat the process corresponding unmatched left and right sub-strings. The complexity of process is reduced as the sequences to be handled are smaller compared to those in earlier step. Common string is added in the super-sequence.

Finding matching sub-string in both the strings is well-defined. Repeating process recursively is with reduced complexity, hence recursive algorithm is feasible. The basic process of having nothing in common (either one of the strings is empty or both the strings have nothing in common) is atomic process which does not result in repeating the process.

2.1 Correctness of Algorithm

If we consider only matching as the measure (not considering penalty for gaps and mismatches), this algorithm give correct result. Proof of the same is in Appendix using Principle of Mathematical Induction.

However, if we wish to have negative score for mismatches and gaps then this algorithm does not give optimal score. It is illustrated by the following example having score of +1 for matching and -1 for each of mismatch and gap.

Sequences:

gaaa

cccg

Optimal Score: -4 (Each position is having mismatch)

Result given by algorithm:

^^^gaaa

cccg^^^

Score: -5 (-6 for gaps and +1 for one matching position).

To overcome this problem, we are tweaking this algorithm later in section Acceptability of matched substring.

2.2 Atomic Matching Process

The atomic process is comparing both the strings and finding largest common sub-string. Outcome of the process does not change if we exchange the strings. So, we can safely assume that first string is not longer than second string. We try to find full first string (trivial sub-string of first string) in second string. If it is found in second string then matching sub-string is found. If no matching takes place then we reduce length by 1. There are 2 sub-strings of the length of our interest. First sub-string starts from first position and second sub-string from second position. As general case, for desired length L, we can build a slider on first string with starting position 1, 2, 3,…, N+1-L. The process starts with full length and goes on reducing length by 1 in each iteration. For selected length, we repeat the process for sub-strings given by slider. In the process, whenever the sub-string is found in second string, the process is ended and matching sub-string and other related information like starting positions of matched sub-string in both the strings are returned as result. When slider length becomes 0, the process is ended with no matching sub-string.

Number of sub-strings of length L created by slider is N + 1 - L where N is length of first string. Number of sub-strings to be searched in second string is 1, 2, 3, …, N for lengths N, N -1, N-2,…,1. The process is ended if matching sub-string is found. The best case for process is 1 iteration (full string match) and worst case is N(N+ 1)/2 iterations (no matching sub-string).

2.3 Acceptability of matched substring

For given sequences A and B of lengths m and n respectively, at least abs (m-n) gaps would occur. If we ignore matched substring, min (m, n) mismatches would be there. If we do not consider matching, the score would be abs (m-n)g + min (m,n)x where g is score for gap and x is score for mismatch. On the other hand, if we consider matched substring (of length k) leaving residual substrings of lengths m1 and n1 (left substrings) and m2 and n2 (right substrings) then possible score would be ky + min (m1,n1)x + abs (m1-n1)g+min (m2,n2) + abs (m2-n2)g

where v is score for matching of element. We accept the matching if there is improvement in the score otherwise we ignore the substring matching.

The cases where a substring from far left of one sequence is matched with far right of another sequence, there would be big gaps between lengths of residual substrings (left with left and right with right) which would result in big insertion penalty. To avoid deterioration in score, we ignore such matching.

2.4 Micro Optimization

For first iteration, matching sub-string can be of any length. However, once a matching sub-string is found, matched length for subsequent search for left and right sub-strings cannot be longer than matched length in parent search. Had there been any longer matching sub-string in left or right remaining strings, it would have matched in earlier iteration. In our example, first largest matching sub-string has length of 7 characters. For left and right sub-string matching cannot have any sub-string larger than 7 characters long. Subsequent recursive process of the searching largest sub-string should use the fact that searching should not be attempted for length larger than matched string length in parent process.

In our example, in first step, largest matching sub-string is of length 7. Left side unmatched sub-strings are 1 and 2 respectively and length of right side unmatched sub-strings are 26 for both of them. Right side sub-string cannot have matched sub-string of length greater than 7. There is no point in trying sub-string lengths of 26, 25, 24, …, 8 as sub-strings of these length were already tried in main string process and matching was unsuccessful. By not trying larger lengths, we are avoiding 1 + 2 + …+ 19 search operations.

This results in a very effective reduction of processing time.

2.5 Estimating conservative negative score

Before starting a search in string of length l in two strings of length m and n assuming that there is no matching substring of length greater than l, the following things are obvious:

- At least int (min(m,n)/(l+1)) mismatches if abs(m-n) < min(m,n)/l – can be easily proved by principle of pigeon hole
- Abs(m-n) gaps would be there

The negative score described in above two points (gaps and mismatches), if we find that if this negative score even with optimistic matching score for other positions can result in non-satisfactory matching score then we can abort the process without doing further process.

Interesting thing to note here is that if substring of larger length is matched, complexity for further process is reduced (because (a2 + b2) < (a+b)2 for a, b > 0). If we do not find matching sub-strings even for relatively small lengths, conservative mismatch score is bigger (in absolute value) which results in earlier identification significantly different sequences.

In actual practice, sequence comparison gives result in much lesser iterations.

2.6 Macro Optimization

Usually a DNA sequence is matched with large number sequences in a database (of sequences). Before matching using this algorithm, which is time consuming, preliminary elimination should be done using the following checks.

- Length of sequence from database should be checked for range. For example, the sequence to be compared with sequences in database has length of 200 and if we want to have target matching of 20% then sequences of length outside 160 and 240 should be discarded without doing process using any algorithm.
- Sequences consist of only 4 pre-defined base elements, so frequency distribution of base elements can be quickly done (very small processing time) and this distribution is common for the main sequence and the sequences in the database can be extended to have this information as well. This optimization can be done with DNA sequences only. For other sequences where either number of possible base elements are large or are not pre-defined then this optimization is not practical. Moreover, for better performance, it expects frequency distribution in the database itself.

If situation demands, we can put constraint on minimum matched length. For example, first (largest) substring matched in the example is 7. If we wish to put a constraint on minimum length of the largest matching substring of 10 then we try slider of length 10 first and do the full process only if a matching sub-string is found.

3. Currently Available Tools

We have considered global sequence matching. One excellent algorithm by Needleman and Wunsch is very popular [5]. In this algorithm grid is prepared with one sequence in horizontal direction and another sequence in vertical direction with an extra row and column with initial values of 0, g, 2g, 3g, … where g is score for gap. Inserts are represented by horizontal and vertical directions whereas comparison (match or mismatch) is considered by diagonal direction. Possible values of score for the position is computed as maximum of positional value (gap for horizontal or vertical direction and match or mismatch score for diagonal direction), The direction(s) of getting maximum score is remembered for trace back. Once grid is populated, trace back from bottom -right to left-top is done. The direction of trace back gets translated into gap or direct comparison.

This algorithm (Needleman and Winch) [5] gives very good results. Important things to note here is that trace back is not unique in all the cases. Sometimes different matching are possible and they may give the same result. For example, sequences ag and ga may give matching as (ag^ and ^ga) or (^ag and ga^); boath giving 1 matching position and two gaps.

Another important point is that the basic algorithm (Needleman and Wunsch) is data neutral. The process time depends only on the length of both the sequences. Best and worst processing time does not change.

Another important type of sequence matching is local sequence matching, Algorithm given by Smith and Waterman [6] is similar to Neeleman and Wunsch algorithm [5] with a small change. At no cell in grid, negative value is populated. Extra row and column has 0 in all the positions and sequence values populated are set to zero

if the maximum from all there directions is negative. Once data is populated in the grid, trace back is done from maximum value to zero. This gives matching of parts of both the sequences resulting in good matching.

The recursive algorithm described here is not meant for this type of matching.

For searching matching sequences in database, tools like BLAST [7] [8] and FASTA [9] [10] use heuristic algorithms which are not very accurate but they give very good approximation.

A very useful resource for DNA Sequence analysis is website of National Center for Biotechnology Information [11].

Some of popular tools for DNA sequence matching are listed below

- BLAST - Basic Local Alignment Search Tool [7] [8]
- FASTA [9] [10]
- DNA Baser [12]
- Matlab - Bioinformatics Toolbox - Sequence Analysis [13]
- R - msa An R Package for Multiple Sequence Alignment [14]
- Mathematica - GnomeData and Sequence Alignment and Comparison [15]
- Python - Packages like HTSeq [16], repDNA [17] and TAMO [18]

4. Other Application

The algorithm described here can be used for general search term matching. This algorithm has strong positional matching. In general search operation, generally order of phrases is interchangeable. For example, we may like to have 'sequence matching' and 'matching of sequence' as good match. The algorithm can be suitably modified for such applications. The complexity for order neutral search string is less compared to ordered sub-string search like DNA sequence matching. For search string, corresponding both left and right the unmatched sub-strings should be concatenated and process should be repeated. This results in non-recursive iterative process. Only constraint should be put that initial matched length should be reasonably large. In the example of 'sequence marching' and 'matching of sequence', first largest matched sub-string is 'sequence' leaving 'matching' and 'matching of' as unmatched strings. Next iteration would match 'matching' thus as the outcome is good match between both the strings. If we do not put constraint of initial string length matching then 'detail' and 'dilute' may get matched which we may not like to happen. So, first match should have certain minimum length constraint.

Another interesting possibility is to use this algorithm for patterns in decimal representation of irrational numbers like π and e. Decimal representation of irrational number of our interest can be stored as a sequence. We can have slider with reasonably big size (say 100) and start from first position. We check existence of the same in remaining part of the sequence (where these 100 characters end) check if the same string occurs there. If succeed, we try to find for more subsequent occurrences. If we do not get good matches, we can move the slider to the right by one position and repeat the process. If we cannot find any matching sub-strings after reasonably big number of shifts, we try by reducing search string length by 1. We go on repeating the process for each length. We may get many matches for single digit sub-strings as all the digits occur in the representation. This

sounds trivial but it would be interesting to know if occurrences of any particular digit form any pattern like positions are in arithmetic or geometric sequence.

5. Conclusion

Though sequence matching solutions are available, understanding this algorithm can be useful in using other tools as well. In certain peculiar situations where custom solution is required, software can be developed using this algorithm. The algorithm can be coded in popular languages such as Java, C-sharp, C, C++ and even VBA for Microsoft Office Suite. Any programming language supporting recursion and string handling can be used to develop the application using this application.

Acknowledgement: The authors thank DST-FIST file # MSI-097 for technical support to the department of Mathematics.

Appendix

Proof of correctness of the algorithm for matching of sequence matching without any penalty for mismatches and gaps is given below.

Statement:

When we match a sequence of length n with another given sequence, the recursive algorithm described in this paper gives optimum result (matching with maximum length).

Proof:

We name the given sequence as master sequence and the sequence to be compared with master sequence as data sequence.

We prove that the statement for n=1.

For n=1, data sequence has only one character. For this sequence, there are two possibilities. The data sequence character is there in the master sequence or the same is not there in the master sequence. The correct outcome is 1 character matching is it is there else 0 characters matching. The algorithm gives the same result.

The algorithm gives correct result in this case.

Algorithm gives correct result for n=1.

Let us assume that the algorithm gives correct length for length = k.

Consider data sequence $D=d_1d_2\ldots d_kd_{k+1}$.

By out Mathematical Induction assumption, algorithm gives correct result for sequence $D'=d_1d_2\ldots d_k$.

Let us say matched length for D' is m. Before proceeding further we understand what happens when data sequence is extended or truncated.

If data sequence is extended by adding elements in any direction, matched sub-string of the original sequence still remains matched in extended sequence (there could be extra matching elements). As a result, matched sequence length for sequence D (extended sequence) cannot be smaller than m which is matched length for sequence D' (the original sequence).

Similarly, when a data sequence is matched with master sequence and a non-matching characters is removed from the data sequence, matching result does not change. If we remove a matching character from data sequence, matching result reduces by one character only.

Using the facts described above, we claim that matched length for sequence D can be either m or m+1 where m is the matched length for sequence D'.

After matching the sequence D' with master sequence, the process ends up with residual right substrings from both master sequence and data sequence D' having no common sub-sequence (not even single character).

The last character d_{k+1} of sequence D has to match (if at all it matches) in the right residue of the master sequence. If it is matched with an element before (on left side) last matched sub-string in master string, it will leave at least the last matched sub-string unmatched. So, the last character of data sequence has to be searched in right residue of master sequence.

The residual substrings can be of zero length (Null strings) or can be of non-zero length (Non-null strings). The table below has all the possibilities of residue string lengths:

		Master Right Residue	
		Null	Not Null
Data Right Residue	Null	Both Exhausted	Only Master Exhausted
	Not Null	Only Data Exhausted	Nothing Common

If both are exhausted, there is no scope for d_{k+1} to match. In this case, algorithm gives matched length as the same given for D'.

If only master is exhausted, again there is no scope for d_{k+1} to match. In this case, the algorithm gives matched length the same as for D' as in the previous case.

If only data is exhausted, character d_{k+1} is searched in master right residue. If d_{k+1} is the first character in right residue of master, the character would have been matched when the right most substring of D' was matched giving matched length as m+1. In other case, the last character d_{k+1} is searched in master right residue. If the character is found, matched length turns out to be m+1. If the character is not found then matched length is m.

If nothing is common then there is no common element in both the sequences. Only possibility left is character d_{k+1} could be there in the right master residue. If the character is matched then matched length is given as m+1. In other case, If the character is not found then matched length is m.

Thus, in all the cases, algorithm gives correct length of m or m+1 correctly for sequence D

References

[1] "DNA Sequencing," [Online]. Available: https://en.wikipedia.org/wiki/DNA_sequencing.

[2] "Random DNA Sequence Generator," [Online]. Available: http://www.faculty.ucr.edu/~mmaduro/random.htm.

[3] G. Tejas, "Comparison of Sequence Alignment Algorithms," *Cornerstone: A Collection of Scholarly and Creative Works for Minnesota State University,* vol. 4, no. 6, pp. 1-7, 2004.

[4] "Sequence Alignment, Mutual Information, and Dissimilarity Measures for Constructing Phylogenies," *PLOS (The Public Library of Science),* vol. 6, no. 1, pp. 1-11, 4 January 2011.

[5] "Needleman–Wunsch algorithm," [Online]. Available: https://en.wikipedia.org/wiki/Needleman%E2%80%93Wunsch_algorithm.

[6] "Smith–Waterman algorithm," [Online]. Available: https://en.wikipedia.org/wiki/Smith%E2%80%93Waterman_algorithm.

[7] "NCBI Blast Homepage," [Online]. Available: https://blast.ncbi.nlm.nih.gov/Blast.cgi.

[8] "BLAST," [Online]. Available: https://en.wikipedia.org/wiki/BLAST.

[9] "FASTA," [Online]. Available: https://www.ebi.ac.uk/Tools/sss/fasta/.

[10] "FASTA," [Online]. Available: https://en.wikipedia.org/wiki/FASTA.

[11] "NCBI Homepage," [Online]. Available: https://www.ncbi.nlm.nih.gov/.

[12] "DNA Baser," [Online]. Available: http://www.dnabaser.com/.

[13] "Help - Aligning Pairs of Sequences," [Online]. Available: https://in.mathworks.com/help/bioinfo/examples/aligning-pairs-of-sequences.html.

[14] "msa: an R package for multiple sequence alignment," [Online]. Available: https://academic.oup.com/bioinformatics/article/31/24/3997/197486.

[15] "Sequence Alignment & Analysis," [Online]. Available: https://www.wolfram.com/mathematica/newin7/content/SequenceAlignmentAndAnalysis/.

[16] "HTSeq—a Python framework to work with high-throughput sequencing data," [Online]. Available: https://www.ncbi.nlm.nih.gov/pmc/articles/PMC4287950/.

[17] "repDNA," [Online]. Available: https://pypi.org/project/repDNA/.

[18] "BioInformatics - TAMO," [Online]. Available: https://academic.oup.com/bioinformatics/article/21/14/3164/266846.