Contemporary Engineering Sciences, Vol. 2, 2009, no. 8, 383 - 390

An Optimal Approach towards Sequence Analysis

A. Priyam

Senior Lecturer, Dept. of Computer Science & Engineering
Birla Institute of Technology, Extension Center Lalpur, Ranchi
amrita.priyam@gmail.com

B. M. Karan

Professor, Head and Dean, Dept. of EEE
Birla Institute of Technology, Mesra, Ranchi.

G. Sahoo

Professor and Head, Dept. of IT and MCA
Birla Institute of Technology, Mesra Ranchi

S. Anwar

Associate Lecturer, Dept. of Computer Science & Engineering
Birla Institute of Technology, Extension Center Lalpur,
Ranchi

Abstract

Dynamic programming is a form of recursion in which intermediate results are saved in a matrix where they can be refereed to later by the program. The paper aims at presenting the calculation of the edit distance between two given strings which includes a cost factor for the different edit operations like copy, insert and delete. The method used in this paper is a form of dynamic programming which includes the formation of a recurrence relation which is then used to represent the problem in a tabular form. In the next step the table is used to trace back to get the possible combinations of edit operations and the one which involves the least cost can be considered the appropriate one.

Keywords: Dynamic Programming, Recurrence Relation, Edit distance.

1 Dynamic Programming

Dynamic programming is a mathematical technique dealing with the optimization of multistage decision process. The word ‘programming’ is used in the mathematical sense of selecting an optimum allocation of resources. It can also be known as recursive optimization. In this a large problem is split into small sub problems each of them involving only a few variables. Ideally, if an optimal solution is obtained for a system, any portion of it must be optional. This is called the “Bellman’s principle of optimality”.

According to this principle “ An optimal policy (set of decisions) has the property that whatever the initial state and decisions are the remaining decisions must constitute an optimal policy with regard to the state resulting from the first decisions” [4].

The challenge of relating computational methods in pattern matching can be considered sequence alignment is fundamental to inferring homology and function. For example, it is generally accepted that if two sequences are in alignment, part or the entire pattern of nucleotides or polypeptides match, then they are similar and may be homologous. Another heuristic is that if the sequence of a protein or other molecule significantly matches the sequence of a protein with a known structure and function, then the molecules may share structure and function. One way to be certain that the solution to a sequence alignment is the best alignment possible is to try every possible alignment, introducing one or more gaps at every position, and computing an alignment score based on aligned character pairs and inexact matches. However, the computational overhead of evaluating all possible alignments of one sequence against another grows exponentially with the length of the two sequences. Dynamic programming is a form of recursion in which intermediate results are saved in a matrix where they can be refereed to later by the program [2].

The comparison can be likened to solving a series of complex mathematical equations with the results of one equation feeding the input of another.
For illustrating the value of dynamic programming in sequence alignment the following function can be considered.

$$\text{Maxvalue} = f\,(X_i, Y_j)$$

In this equation Maxvalue is some function of variable X_i and Y_j, where i and j are indices to the variable.

Values for X and Y are defined in the tree structure.

Maximizing Maxvalue requires evaluating the equation for every combination of i and j.

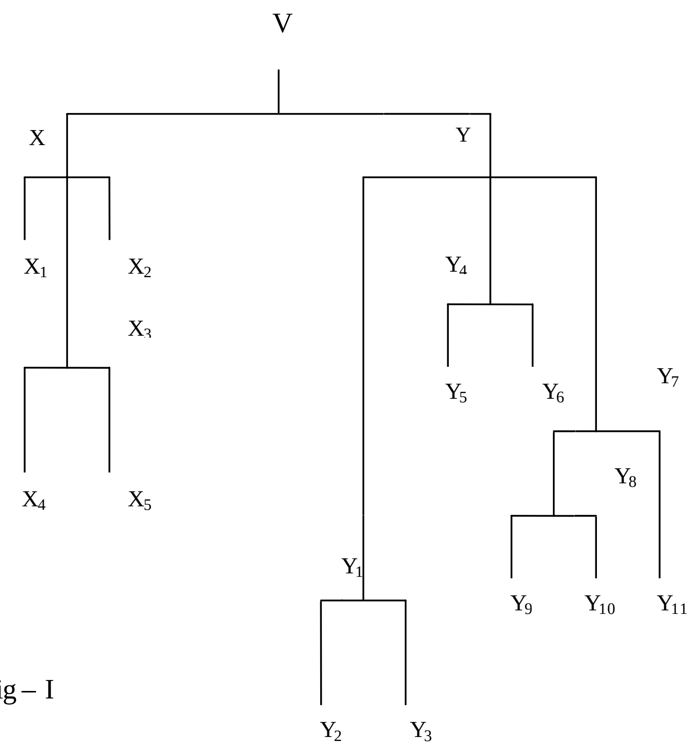

Fig – I

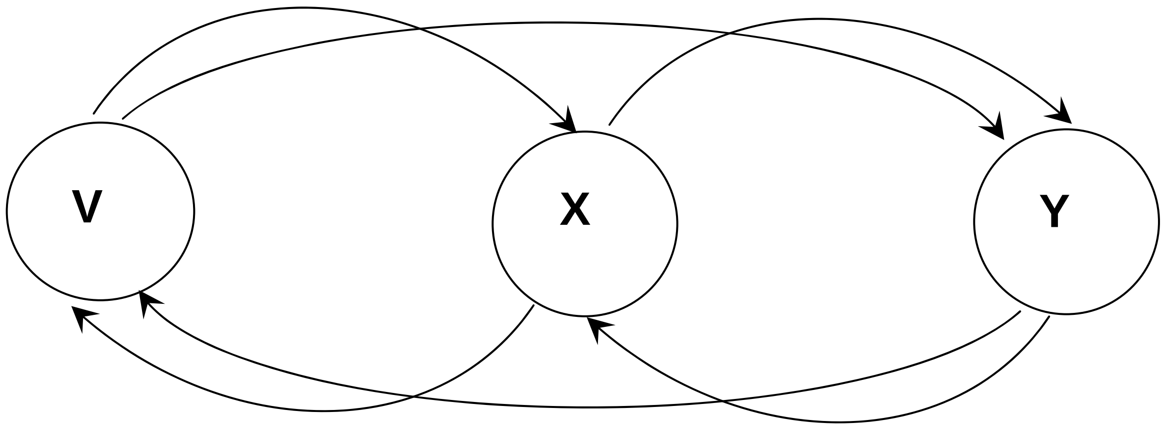

Fig-2

Maximizing Maxvalue requires evaluating the equation for every combination of i and j.

As a case in molecular biology the edit distance between two strings can be considered. Frequently one wants a measure of the difference or distance between two strings for example in evolutionary, structural or functional studies of biological strings. There are several ways to formalize the notion of distance between strings. One common, and simple formalization called edit distance focuses on transforming (or editing) one string into the other by a series of edit operations on individual characters [1].

The permitted edit operations are insertion of a character into the second string, represented by I, the deletion of a character from the first string, represented by D and the copy of a character from the first string to the second string, represented by C.

Based on the central property of an edit graph that any shortest path from start node to the destination node specifies an edit transcript with the minimum number of edit operations. Equivalently any shortest path specifies a global alignment of minimum total weight.

Given two strings S_1 and S_2 of lengths n and m, respectively, a weighted edit graph has (n+1) X (m+1) nodes, each labeled with a distinct pair (i, j) ($0 \leq i \leq n, 0 \leq j \leq m$).

We now turn to the algorithmic question of how to compute, via dynamic programming, the edit distance of two strings, along with the accompanying edit transcript or alignment. From the mathematical standpoint, an alignment and an edit transcript are equivalent ways to describe a relationship between two strings. Specifically, two opposing characters that mismatch in an alignment correspond to a substitution in the equivalent edit transcript a space in an alignment contained in the first string corresponds in the transcript to an insertion of the opposing character into the first string and a space in the second string corresponds to a deletion of the opposing character from the first string.

The specific application of the dynamic programming approach to the edit distance problem can be considered. For two strings S_1 and S_2, D (i, j) is defined to be the edit distance of S_1 [l….i] and S_2 [l…… j].

That is D (i, j) denotes the minimum numbers of edit operations needed to transform the first i characters of S_1into the first j characters of $S_{2.}$

Using this notation, if S_1 has n letters and S_2 has m letters, then the edit distance of S_1 and S_2 is precisely the value D(n, m) [4]. The dynamic programming approach has three essential components –

A) The recurrence relation
B) The tabular computation
C) The trace back

2 The Recurrence Relation

The recurrence relation establishes a recursive relationship between the value of

D(i, j) for i and j both positive, and values of D with index pairs smaller than i, j. When there are no smaller indices the value of D(i, j) must be stated exp D(i, j) explicably in what are called the base conditions for D(i, j). For the edit distance problem the base conditions are D(i, j) = i and D(i, j) = j.

The base condition D(i, j) = i is clearly correct i.e. it gives the number required by the definition of D(i, 0) because the only way to transform the first i characters of S_1 to zero characters of S_2 is to delete all the i characters of $S_{1,}$ similarly, the conditions D(0, j) = j is correct because j characters must be inserted to convert zero characters of S_1 to j characters of $S_{2.}$

The recurrence relation for D (i, j) when both i and j are strictly positive is
D (i, j) = min [D (i, j), j) + l, D (i, j - l) + l, D (i - l, j-l) + t(i, j)],
Where t(i, j) is defined to have value 1 if S_1 (i) $\neq S_2$ (j)
And t(i, j) = 0 if S_1 (i) = S_2 (j)

2.1 Correctness of the general Recurrence

Using the concept of an edit transcript we can establish correctness in the following two lemmas:

Lemma 1
The value of D(i, j) must be D (i, j – l) + 1, D (i – l, j) + 1 Or D (i – l, j - l) + t (i, j) There are no other possibilities.
Proof: Consider an edit transcript for the transformation of S_1 [l….i] to S_2 [l….j] using the minimum no. of edit operations and focus on the last symbol in that transcript. The last symbol must be insertion, deletion or copy { I, D, C}

Case i) If the last symbol is an I then the last edit operation is the insertion of character S_2 (j) onto the end of the (transformed) first string.
If the last symbol in the transcript is I, then D (i, j) = D (i, j – l) + 1
Case ii) Let us consider the last symbol on the transcript is a D In this case the last edit operation is the deletion of S_1 (i) and the symbols in the transcript to the left of that D must specify the minimum no. of edit operations to transform S_1 [l….i – l] to S_2 [l…..j].
By definition, the latter transformation takes D (i – l, j) edit operations. Therefore, if the last symbol in the transcript is D, then D(i, j) = D (i – l, j) + 1
Case iii) Let the last symbol in the transcript is a C**.**
Then S_1 (i) = S_2(j) and D(i, j) = D (i – l, j – l)
Now let us use the variable t (i, j), if S_1 (i) = S_2(j) Then t (i, j) = 0 and if S_1(i) $\neq S_2$(j) then t (i, j) = l

THEOREM 1. When both i and j are strictly positive
D(i, j) = min [D (i – l, j) + l, D (i, j – l) + 1, D (i – l, j – l) + t (i, j)]

Proof: Lemma 1 says that D(i, j) must be equal to one of the three values D (i – l, j) +l, D (i, j – l) + 1 or D (i – l, j – l) + t (i, j).

Lemma 2 says that D(i, j) must be less than or equal to the smallest of those three values. It follows that D(i, j) must therefore be equal to the smallest of those three values. Therefore the theorem has been proved.

2.2 Tabular Composition of the Edit Distance

The second essential component of any dynamic program is to use the recurrence relations to efficiently compute the value D (n, m). The recurrence relations and base conditions for D(i, j) as a recursive computer procedure using a programming language (for ex: JAVA) can be coded.

The top-down recursive approach to evaluate D (n, m) is simple to program but is extremely inefficient for large values of n and m. The problem is that the number of recursive calls grows exponentially with n and m. But there are only (n+1) x (m+1) combinations of i and j. Therefore there are only (n+1) x (m+1) district recursive calls possible. Therefore, the inefficiency of the top down approach is due to a massive number of redundant recursive calls to the procedure. Therefore bottom-up computation is carried out.

Bottom Up Computation

In this approach, first D(i, j) for the smallest possible values for i and j is computed and then values of D(i, j) for increasing values of i and j are computed.

Typically, this computation is organized with a dynamic programming table of size (n+1) x (m+1). The table holds the values of D (i, j) for all the choices of i and j. In the table, the string S_1 corresponds to the vertical axis of the table while string S_2 corresponds to the horizontal axis. As the ranges of i and j begin at zero The table has a zero row and zero column. The row zero and column zero are filled in directly from the base conditions for D(i, j). After that, the remaining n x m sub table is filled in one row at time, in order of increasing i – within each row, the cells are filled in order of increasing j. For example, all the values needed for the computation of D(l, l) are known once D(0, 0), D(l, 0) and D(0, l) have been computed. By extension, the entire table can be filled in one row at a time, in order of increasing i and in each row the values can be computed in order of increasing j. The values in row zero and column zero are already included. They are given directly by the base conditions. Edit distances are filled in one row at a time, and in each row they are filled in from left to right. A cost is assigned to each operation as 5 units to copy, 10 units to insertion and 10 units to deletion.

2.3 The Trace Back

Now once the value of the edit distance has been computed, the question arises of the extraction of the optional edit transcript in this case, the easiest way is to establish pointers in the table as the table values are computed. For example when the value of all (i, j) is computed

i) a pointer is set from cell (i, j) to cell (i, j - l) if D(i, j) = D (i, j - l)+l
ii) a pointer is set from (i, j) to (i - l, j) if D (i, j) = D(i - l, j) + l
iii) A pointer is set from (i, j) to (i - l, j - l) if D(i, j) = D (i - l, j - l) + t (i, j).

This rule is applied to cell in row zero and column zero as well therefore for most objective functions, each cell in row zero points to the cell to its left and each cell in column zero points to the cell just above it. For other cells, if is possible that more than one pointer is set from (i, j).

The advantage of including pointers is to allow easy recovery of an optional edit transcript simply follow any path of pointers from cell (n, m) to cell (0, 0). The edit transcript is recovered from the path by interpreting each horizontal edge in the path from cell (i, j) to cell (i, j - l) as an insertion (I) of character S_2 (j) into S_1 interpreting each vertical edge, from (i, j) to (i - l, j), as a deletion (D) of S_1 (i) from $S_{1;}$ and interpreting each diagonal edge from (i, j) to(i - l, j - l) as a copy (C) if S_1 (i) = S_2 (j).

Alternatively, in terms of aligning S_1 and S_2 each horizontal edge in the path specifies a space inserted into S_1 each vertical edge specifies a space inserted into S_2 and each diagonal edge specifies either a match or a mismatch, depending on the specific characters. The table below shows the edit distances for the transformation of string "SUNSHINE" to string "SUNLIGHT" including the cost factor. The pointers set here are used to indicate a set of possible actions for the string transformation/

D (i, j)		S	U	N	S	H	I	N	E
	0 Null	10 ←	20 ←	30 ←	40 ←	50 ←	60 ←	70 ←	80 ←
S	↑ 10	↖ 5	15 ←	25 ←	↖ 35	45 ←	55 ←	65 ←	75 ←
U	↑ 20	↑ 15	↖ 10	20 ←	30 ←	40 ←	50 ←	60 ←	70 ←
N	↑ 30	↑ 25	↑ 20	↖ 15	25 ←	35 ←	45 ←	↖ 55 ←	65 ←
L	↑ 40	↑ 35	↑ 30	↑ 25	↑ 35 ←	↑ 45 ←	↑ 55 ←	↑ 65 ←	↑ 75 ←
I	↑ 50	↑ 45	↑ 40	↑ 35	↑ 45 ←	↑ 55 ←	↖ 50	60 ←	70 ←
G	↑ 60	↑ 55	↑ 50	↑ 45	↑ 55 ←	↑ 65 ←	↑ 60	↑ 70 ←	↑ 80 ←
H	↑ 70	↑ 65	↑ 60	↑ 55	↑ 65 ←	↖ 60	↑ 70 ←	↑ 80 ←	↑ 90 ←
T	↑ 80	↑ 75	↑ 70	↑ 65	↑ 75 ←	↑ 70	↑ 80 ←	↑ 90 ←	↑ 100 ←

Some of the optional alignments are

1. C,C,C,I,I,I,D,C,I,D,D,D
   S U N _ _ _ S H _ I N E
   S U N L I G _ H T _ _ _

2. C,C,C,D,D,I,C,D,I,I,I,D
   S U N S H _ I N _ _ _ E
   S U N _ _ L I _ G H T _

3. C,C,C,D,D,I,C,D,D,I,I,I
   S U N S H _ I N E _ _ _
   S U N _ _ L I _ _ G H T

If there is more than one pointer from cell (n, m) then a path from (n, m) to (0, 0) can start with either of those pointers. Each of the them is on a path from (n, m) to (0, 0). This property is repeated from any cell encountered. Therefore a trace back path from (n, m) to (0, 0) can start simply by following any pointer out of (n, m) it can then be extended by following any pointer out of any cell encountered. Every cell except (0, 0) has a pointer out of it, so no path from (n, m) can get stuck. Once the dynamic programming table with pointers has been computed an optional edit transcript can be found in o(n + m) time.

3 Conclusion

The method presented above is one of the methods to calculate the edit distances between two given strings. The example taken here are for the two strings "SUNSHINE" and "SUNLIGHT". But the same approach can be extended to apply for actual DNA sequences. The various optional combinations can be studied and the one with the least cost can be considered for further evaluation.

References

[1] D.E Knuth, The Art of programming, Addison Wesley,

[2] G.Nemhauser, Introduction to Dynamic Programming, John Wiley &sons, 2007.

[3] R.A Wagner & M.J Fischer, The String To String Correction Problem, Journal of the ACM21,No.1(1974).168-173.

[4] R.E Bellman & S.E Dreyfus , Applied Dynamic Programming, Princeton University Press.

Received: March, 2009