

## Protein Comparison in Bioinformatics

In this project, the comparison between two strings of alphabetical characters ('proteins') is explored. The objective is to transform a string  $S$  to another string  $T$ , using these operations:

1. Deleting a character
2. Inserting a character
3. Replacing a character with another character

Notation:

Let  $S_i$  and  $T_j$  be the  $i^{th}$  character of  $S$  and  $j^{th}$  character of  $T$  respectively

Let  $S[a, b] = S_a S_{a+1} \dots S_b$  and similarly for  $T$

Let  $D(i, j) = d(S[1, i], T[1, j])$  = edit distance between  $S[1, i]$  and  $T[1, j]$ , where edit distance between two strings is the number of operations to transform the first to the second

To transform  $S[1, i] = S_1 \dots S_i$  into  $T_1 \dots T_j = T[1, j]$ , one of the following cases must be true, since the final character must match:

### Case 1:

$S_i$  is deleted

$$S_1 \dots S_{i-1} S_i \xrightarrow{1 \text{ operation}} S_1 \dots S_{i-1} \xrightarrow{D(i-1, j) \text{ operations}} T_1 \dots T_j$$

$$D(i, j) = 1 + D(i-1, j)$$

### Case 2:

$T_j$  is inserted

$$S_1 \dots S_i \xrightarrow{1 \text{ operation}} S_1 \dots S_i T_j \xrightarrow{D(i, j-1) \text{ operations}} T_1 \dots T_{j-1} T_j$$

$$D(i, j) = 1 + D(i, j-1)$$

### Case 3:

$S_i$  is replaced with  $T_j$  (or unchanged if  $S_i = T_j$ )

If  $S_i = T_j$ :

$$D(i, j) = D(i-1, j-1)$$

If  $S_i \neq T_j$ :

$$S_1 \dots S_{i-1} S_i \xrightarrow{1 \text{ operation}} S_1 \dots S_{i-1} T_j \xrightarrow{D(i-1, j-1) \text{ operations}} T_1 \dots T_{j-1} T_j$$

$$D(i, j) = 1 + D(i-1, j-1)$$

Hence, the minimal number of edits is the minimum of these three expressions:

$$D(i, j) = \min\{D(i-1, j) + 1, D(i, j-1) + 1, D(i-1, j-1) + 1\{S_i \neq T_j\}\}$$

Implementing this in MATLAB:

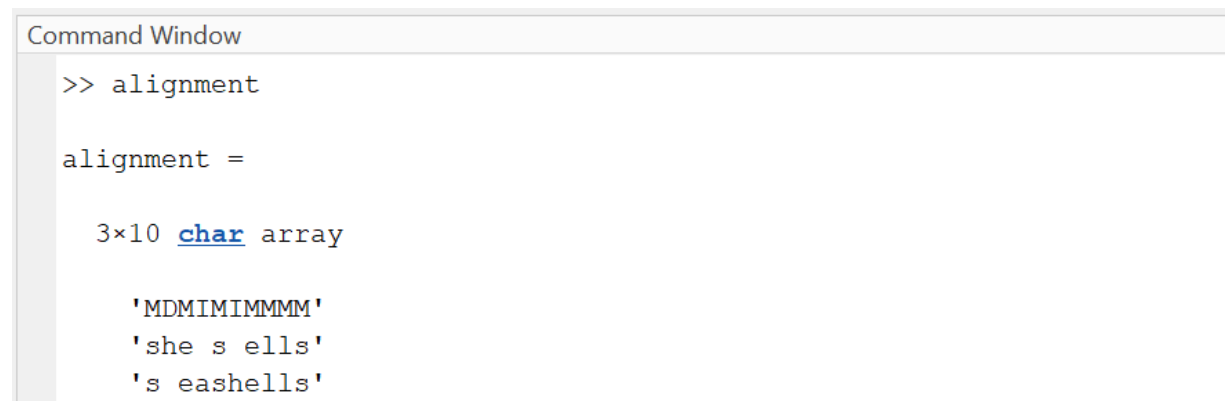
```
function [distance] = editdistance(S,T)
    i = length(S);
    j = length(T);
    if i == 0
        distance = j;
        return
    end
    if j == 0
        distance = i;
        return
    end

    indicator = isequal(S(i), T(j));
    distance = min([editdistance(S(1:i-1),T)+1 editdistance(S,T(1:j-1))+1
editdistance(S(1:i-1),T(1:j-1))+1-indicator])
end
```

**Figure 1: editdistance.m**

The function editdistance.m is shown above in figure 1, which is called recursively until a base case is reached, i.e.  $i = 0$  or  $j = 0$ . This function is tested with the input  $S = \text{shesells}$  and  $T = \text{seashells}$ , which gives the correct result of 3.

At every recursion, the length of S or T (or both) are reduced by 1, so it reaches the base case in at most  $m+n$  steps. editdistance.m is called 3 times at every recursion, which means that the complexity is  $O(3^{m+n})$ . This gets very large quickly and is an inefficient solution. To optimise this, the dynamic programming approach is taken, which stores intermediate results and avoids repeated calculations. The program is also further improved to show the alignment steps, rather than the resulting distance only, shown in figure 3. This gives the following result (figure 2):



```
Command Window

>> alignment

alignment =

3x10 char array

'MDMIMIMMMM'
'she s ells'
's eashells'
```

**Figure 2: alignment for test case**

The first string of letters represent the alignment, where:

M = match

D = delete

I = insert

R = remove (not shown in this example)

```

function [distance, transcript] = editdistancematrix(S,T)
    i = length(S);
    j = length(T);
    distancematrix = zeros(i+1, j+1);
    distancematrix(1,:) = 0:j;
    distancematrix(:,1) = 0:i;
    alignmentmatrix = repmat({'0'}, [i+1 j+1]);
    for index = 1:j+1
        alignmentmatrix(1, index) = {repmat('2', [1,index-1])};
    end
    for index = 1:i+1
        alignmentmatrix(index, 1) = {repmat('1', [1,index-1])};
    end

    for index1 = 2:i+1
        for index2 = 2:j+1
            if S(index1-1) == T(index2-1)
                distancematrix(index1, index2) = distancematrix(index1-1, index2-1);
                alignmentmatrix(index1, index2) = {append(alignmentmatrix{index1-1, index2-1}, '4')};
            else
                [a, b] = min(1+[distancematrix(index1-1, index2) distancematrix(index1, index2-1)]);
                distancematrix(index1, index2) = a;
                if b == 1
                    alignmentmatrix(index1, index2) = {append(alignmentmatrix{index1-1, index2}, '1')};
                elseif b == 2
                    alignmentmatrix(index1, index2) = {append(alignmentmatrix{index1, index2-1}, '2')};
                else
                    alignmentmatrix(index1, index2) = {append(alignmentmatrix{index1-1, index2-1}, '3')};
                end
            end
        end
    end
    distance = distancematrix(i+1, j+1);
    transcript = alignmentmatrix(i+1, j+1);
end

```

**Figure 3: more efficient editdistancematrix.m (using dynamic programming)**

This new program makes it possible to compute the distance and alignment between much longer strings, as its complexity is now only  $O(mn)$ .

Next, we consider an alternative scoring method that accounts for the biological differences between different strings of proteins, which adjusts for the probabilities of different mutations. The BLOSUM62 substitution matrix is used to calculate the value of replacing  $S_i$  with  $T_j$ , denoted by  $s(S_i, T_j)$ . This scoring method attempts to maximise a value measure rather than minimising distance, and also allows deletion or insertion of large chunks of characters.

Notation:

Let  $w(l) < 0, l \geq 1$  be the score of deleting or inserting a sequence of length  $l$

Let  $V_{gap}(i, j) = v_{gap}(S[1, i], T[1, j])$  where  $v_{gap}$  is the gap-weighted score

It can be shown that  $V_{gap}(i, j) = \max\{E(i, j), F(i, j), G(i, j)\}$ , where

$$E(i, j) = \max_{0 \leq k \leq j-1} \{V_{gap}(i, k) + w(j - k)\}$$

$$F(i, j) = \max_{0 \leq k \leq i-1} \{V_{gap}(k, j) + w(i - k)\}$$

$$G(i, j) = V_{gap}(i - 1, j - 1) + s(S_i, T_j)$$

Consider the simpler case where  $w(l) = u$  for some constant  $u$ . The  $v_{gap}$  value is found using the following logic:

Let  $V_{gap}$  matrix be the following: (m+1 by n+1)

0	u	u	u	....				
u								
u								
u								
⋮								

j

i

Figure 4: color-coded  $V_{gap}$  matrix

To find the gap-weighted score, the entries of the  $V_{gap}$  matrix have to be calculated iteratively from the top left corner until it reaches the bottom right corner. The iteration is such that when computing an entry, all the entries above it and to the left are known.

In figure 7, suppose we want to compute  $V_{gap}(i, j)$ , the entry colored in red.

Since  $w(l)$  constant:

1. Finding  $E(i, j)$  = finding the max of the green entries, then adding  $u$
2. Finding  $F(i, j)$  = finding the max of the blue entries, then adding  $u$
3. Finding  $G(i, j)$  = adding a known constant to the yellow entry

To reduce unnecessary computations in step 1 and 2, introduce four new matrices,  $rowMax$ ,  $colMax$ ,  $rowMaxIndex$  and  $colMaxIndex$  such that:

- $rowMax(i, j) = \max_{k \leq j-1} V_{gap}(i, k)$  (max of the green entries)
- $colMax(i, j) = \max_{k \leq i-1} V_{gap}(k, j)$  (max of the blue entries)
- $rowMaxIndex, colMaxIndex$  store the location of the max green/blue entry

This is more efficient, because instead of finding the max of  $(i - 1) + (j - 1)$  values every iteration (the green and blue entries), we only have to find the max of 5 values:

1. Max of first  $j - 2$  green squares:  $rowMax(i, j - 1) + u$
2.  $(j - 1)^{th}$  green square:  $V_{gap}(i, j - 1) + u$
3. Max of first  $i - 2$  blue squares:  $colMax(i - 1, j) + u$
4.  $(i - 1)^{th}$  blue square:  $V_{gap}(i - 1, j) + u$
5. Yellow square + score function:  $V_{gap}(i - 1, j - 1) + s(S_i, T_j)$

Then, the  $(i, j)$  entry is updated for all the relevant matrices and the iteration continues.

This means that every iteration can be done in  $O(1)$  operations. Hence the algorithm has total complexity  $O(mn)$ .

The algorithm is implemented below using the boundary conditions  $V(0, k) = V(k, 0) = u \quad \forall k \neq 0$

```
function [score, transcript, V] = gapweighted(S,T,blosum,letters,u)
    i = length(S);
    j = length(T);
    %Initialising matrices
    V = zeros(i+1, j+1);
    rowMax = zeros(i+1, j+1);
    colMax = zeros(i+1, j+1);
    rowMaxEntry = zeros(i+1, j+1);
    colMaxEntry = zeros(i+1, j+1);
    V(1,2:end) = u;
    V(2:end,1) = u;
    rowMax(1,:) = 0;
    rowMax(:,1) = -inf;
    colMax(1,:) = -inf;
    colMax(:,1) = 0;
    rowMaxEntry(1,:) = 1;
    rowMaxEntry(:,1) = -inf;
    colMaxEntry(1,:) = -inf;
    colMaxEntry(:,1) = 1;
    alignmentmatrix = repmat({'0'}, [i+1 j+1]);
    for index = 1:j+1
        alignmentmatrix(1, index) = {repmat('2', [1,index-1])};
    end
    for index = 1:i+1
        alignmentmatrix(index, 1) = {repmat('1', [1,index-1])};
    end

    %Iterating
    for index1 = 2:i+1
        for index2 = 2:j+1
            [E, temp1] = max(u+[rowMax(index1,index2-1) V(index1,index2-1)]);
            [F, temp2] = max(u+[colMax(index1-1,index2) V(index1-1,index2)]);
            letterIndex1 = letters==S(index1-1);
            letterIndex2 = letters==T(index2-1);
            G = V(index1-1, index2-1)+blosum(letterIndex1, letterIndex2);
            [a,b] = max([E F G]);
            V(index1, index2) = a;
            if b == 1
                if temp1 == 1
                    maxIndex = rowMaxEntry(index1, index2-1);
                    diff = index2 - maxIndex;
                    alignmentmatrix(index1, index2) = {append(alignmentmatrix{index1, maxIndex}, repmat('2',
[1,diff]))};
                else
                    alignmentmatrix(index1, index2) = {append(alignmentmatrix{index1, index2-1}, '2')};
                end
            elseif b == 2
                if temp2 == 1
                    maxIndex = colMaxEntry(index1-1, index2);
                    diff = index1 - maxIndex;
                    alignmentmatrix(index1, index2) = {append(alignmentmatrix{maxIndex, index2}, repmat('1',
[1,diff]))};
                else
                    alignmentmatrix(index1, index2) = {append(alignmentmatrix{index1-1, index2}, '1')};
                end
            else
                if S(index1-1) == T(index2-1)
                    alignmentmatrix(index1, index2) = {append(alignmentmatrix{index1-1, index2-1}, '4')};
                else
                    alignmentmatrix(index1, index2) = {append(alignmentmatrix{index1-1, index2-1}, '3')};
                end
            end
            %Updating rowMaxEntry and colMaxEntry
            if temp1 == 1
                rowMaxEntry(index1, index2) = rowMaxEntry(index1, index2-1);
                rowMax(index1, index2) = rowMax(index1, index2-1);
            else
                rowMaxEntry(index1, index2) = index2-1;
                rowMax(index1, index2) = V(index1, index2-1);
            end
            if temp2 == 1
                colMaxEntry(index1, index2) = colMaxEntry(index1-1, index2);
                colMax(index1, index2) = colMax(index1-1, index2);
            else
                colMaxEntry(index1, index2) = index1-1;
                colMax(index1, index2) = V(index1-1, index2);
            end
        end
    end
    score = V(i+1, j+1);
    transcript = alignmentmatrix(i+1, j+1);
end
```

**Figure 5: gapweighted.m**

The function is tested with  $u = 12$  and the following proteins, which are both keratin structures in humans.

```
C =
'MTSDCSSTHCSPESCGTASGCAPASSCSVETACLPGTCATSRCTPSFLSRSGLTGCLLPCYFTGSCNSPCLVGNCWCE
DGVFTSNEKETMQFLNDRLASYLEKVRSLEETNAELESRIQEQQEQDIPMVCPTYQRYFNTIEDLQQKILCTKAENSRLAVQ
LDNCKLATDDFKSKYESELSLRQLLEADISSLHGILEELTLCKSDLEAHVESLKEDLLCLKKNHEEEVNLLREQLGDRLSVE
LDTAPTLDLNRVLDEMRCQCETVLANNRREAEEWLAVQTEELNQQQLSSAEQLQGCQMEILELKRTASALEIELQAQQLTE
SLECTVAETEAQYSSQLAQIQCLIDNLENQLAEIRCDLERQNEQYQVLLDVKARLEGEINTYWGLLSEDSRLSCSPCSTTC
TSSNTCEPCSAVICTVENCCL';
D =
'MPYNFCLPSLSCRTSCSSRPCVPPSCHSCTLPGACNIPANVSNCNWFCEGSFNGSEKETMQFLNDRLASYLEKVRQLERDN
AELENLIRERSQQQEPLLCPSYQSYFKTIEELQQKILCTKSENARLVVQIDNAKLAADDFRTKYQTELSLRQLVESDINGLR
RILDELTLCKSDLEAQVESLKEELLCLKSNHEQEVNTRCQLGDRLNVEVDAAPTVDLNRVLNETRSQYEALVETNRREVEQ
WFTTQTEELNKQVSSSEQLQSYQAEIIELRRTVNALEIELQAQHNLRDSLENTLTSEARYSSQLSQVQSLITNVESQLAE
IRSDLERQNEQYQVLLDVRARLECEINTYRSLLESEDCLNLPSPNCPATTNACSKPIGPCLSNPCTSCVPPAPCTPCAPRPRCG
PCNSFVR';
```

```
[score, transcript] = gapweighted(C,D,blosum,letters,-12);
```

This gives a score of 1236 and the first 50 alignment steps:

```
>> alignment(:,length(alignment)-49:end)

ans =

3×50 char array

'MIIIIIIIIIIIIIIIIIIIIIIIIIIIIIMRRRRRRMRMMRRRMDDDDDDDDR'
'T                                     CTSSNTCEPCSAVICTVENCCL'
'TNACSKPIGPCLSNPCTSCVPPAPCTPCAPRPRCGPCNSFV          R'
```

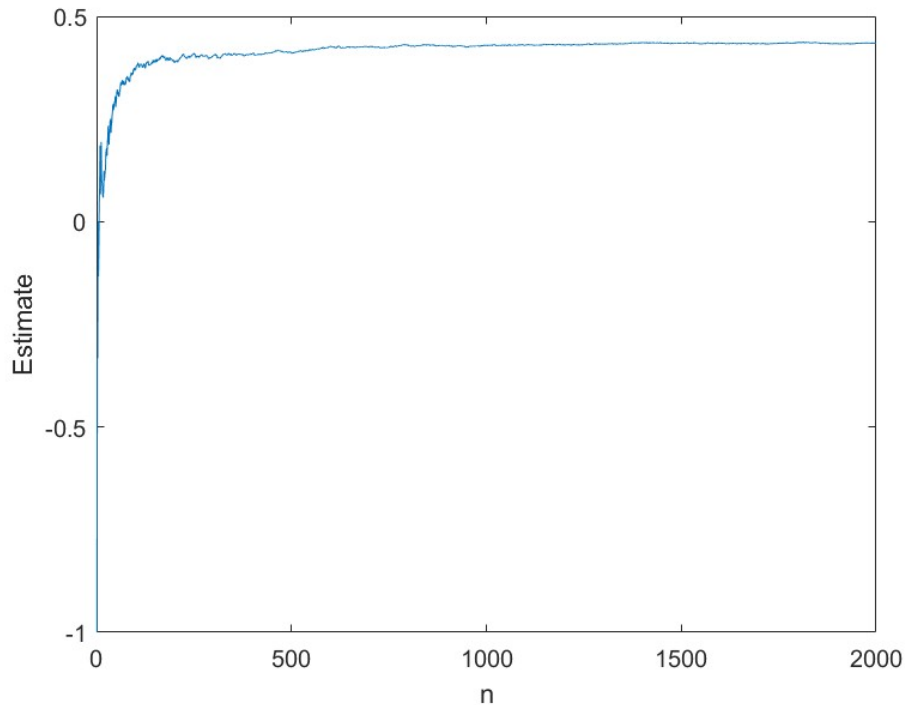
To determine if the score  $v_{gap}(S, T)$  is statistically significant, we must compute the expected score between two unrelated proteins, assumed to be random. Consider the simple case where there are only 2 letters in the alphabet, a and b, with  $s(a, a) = s(b, b) = 1$  and  $s(a, b) = s(b, a) = -1$ . Suppose  $U^n$  and  $V^n$  are independent and identically distributed random proteins with length  $n$ ,  $\mathbb{P}(U_i = a) = 0.5$  and  $\mathbb{P}(U_i = b) = 0.5$  for all  $i$ . We assume the score of inserting or deleting a chunk of protein to be  $u = -3$ . The expected gap weighted score,  $\mathbb{E}[v_{gap}(U^n, V^n)]$  is then estimated by the following code, `estimate.m`, shown in figure 6.

```
function [estimate] = estimate(n,u,k)
    total = zeros(1,n);
    estimate = zeros(1,n);
    for i = 1:k
        Un = char(randi([97 98],1,n));
        Vn = char(randi([97 98],1,n));
        [~, ~, V] = gapweighted(Un,Vn,[1 -1; -1 1],['a' 'b'],u);
        scores = diag(V);
        total = total + scores(2:end);
    end
    for j = 1:n
        estimate(j) = total(j)/(j*k);
    end
end
```

Figure 6: `estimate.m`

The function generates two strings,  $U^n$  and  $V^n$  and calculates the score, then repeats this  $k$  times to estimate  $\mathbb{E}[v_{gap}(U^n, V^n)]$  for a fixed  $n$ . This program has complexity  $O(n^2k)$ .

This function is first called with the parameters  $n = 2000, k = 3$ , which yields the following result:



*Figure 7: plot of estimate against  $n$*

As seen from the plot above (figure 7), the estimate does not change much past  $n > 1500$ . The average of the last 500 values is then taken to be the estimate of the limit. The max and min value past  $n > 1500$  differ by only 0.004, which is a relative fluctuation of less than 1%. This suggests that the estimate of the limit shown below is fairly accurate.

```
>> mean(limit(1501:2000))
```

```
ans =
```

```
0.4321
```

When comparing proteins, it is useful to identify regions that are highly similar. We attempt to find

$v_{sub}(S, T) = \max \{v(S', T') : S' \text{ substring of } S, T' \text{ substring of } T\}$ , where  $v$  is the alignment score.

Notation:

Let  $s(-, a) = s(a, -) < 0$  for the score of an insertion or deletion

$v_{sfx}(S, T) = \max \{v(S', T') : S' \text{ suffix of } S, T' \text{ suffix of } T\}$

Want to show:  $v_{sub}(S, T) = \max \{v_{sfx}(S', T') : S' \text{ prefix of } S, T' \text{ prefix of } T\}$

Suppose  $v_{sub}(S, T) = v(S[a, b], T[c, d])$

Then  $v_{sfx}(S[1, b], T[1, d]) = \max_{x \leq b, y \leq d} \{v(S[x, b], T[y, d])\} \geq v_{sub}(S, T)$

$\Rightarrow \max_{b', d'} \{v_{sfx}(S[1, b'], T[1, d'])\} \geq v_{sub}(S, T)$

$\Rightarrow RHS \geq LHS$

Assume  $RHS > LHS$ :

Then  $v_{sfx}(S[1, x], T[1, y]) > v_{sub}(S, T)$  for some  $x, y$

$\Rightarrow v(S[p, x], T[q, y]) > v_{sub}(S, T)$  for some  $p \leq x, q \leq y$  by definition of  $v_{sfx}$

$\Rightarrow$  Contradiction, by definition of  $v_{sub}(S, T)$

Hence  $RHS = LHS$ .

From previous parts, we have

$V(i, j) = \max \{V(i-1, j-1) + s(S_i, T_j), V(i-1, j) + s(S_i, -), V(i, j-1) + s(-, T_j)\}$  (\*)

where  $V(i, j) = v(S[1, i], T[1, j])$

$$\begin{aligned}
 v_{sfx}(i, j) &= \max_{x \leq i, y \leq j} \{v(S[x, i], T[y, j])\} \\
 &= \max \begin{cases} \max_{x < i, y < j} \{v(S[x, i], T[y, j])\} \\ \max_{x=i, y < j} \{v(S[x, i], T[y, j])\} = s(-, T_j) < 0 \\ \max_{x < i, y=j} \{v(S[x, i], T[y, j])\} = s(S_i, -) < 0 \\ \max_{x=i, y=j} \{v(S[x, i], T[y, j])\} = 0 \end{cases} \\
 &= \max \begin{cases} 0 \\ \max_{x \leq i-1, y \leq j-1} \{v(S[x, i], T[y, j])\} \end{cases} \\
 &= \max \begin{cases} 0 \\ \max_{x \leq i-1, y \leq j-1} \{v(S[x, i-1], T[y, j-1]) + s(S_i, T_j)\} \\ \max_{x \leq i-1, y \leq j-1} \{v(S[x, i-1], T[y, j]) + s(S_i, -)\} \\ \max_{x \leq i-1, y \leq j-1} \{v(S[x, i], T[y, j-1]) + s(-, T_j)\} \end{cases} \quad \text{by (*)} \\
 &= \max \begin{cases} 0 \\ \max_{x \leq i-1, y \leq j-1} \{v(S[x, i-1], T[y, j-1])\} + s(S_i, T_j) \\ \max_{x \leq i-1, y \leq j} \{v(S[x, i-1], T[y, j])\} + s(S_i, -) \\ \max_{x \leq i, y \leq j-1} \{v(S[x, i], T[y, j-1])\} + s(-, T_j) \end{cases} \quad \text{(**)}
 \end{aligned}$$



$$= \max \begin{cases} 0 \\ V_{sfx}(i-1, j-1) + s(S_i, T_j) \\ V_{sfx}(i-1, j) + s(S_i, -) \\ V_{sfx}(i, j-1) + s(-, T_j) \end{cases}$$

For (\*\*), the max in the third case is taken over  $x \leq i-1, y \leq j$  instead of  $x \leq i-1, y \leq j-1$ , since  $y = j \Rightarrow v(S[x, i-1], T[y, j]) + s(S_i, -) = v(S[x, i-1], "") + s(S_i, -) < 0$  (similarly for the fourth case), so it gives the same result.

Boundary conditions:

$$v_{sfx}(S[1, i], "") \begin{cases} < 0 \text{ for all } i > 1 \\ = 0 \text{ for } i = 1 \end{cases}$$

$$\Rightarrow V_{sfx}(i, 0) = 0$$

$$\text{Similarly, } V_{sfx}(0, j) = 0$$

To find  $v_{sub}$ , a similar method is used. A matrix is used to store the values of  $V_{sfx}(i, j)$ , and the entries of the matrix are calculated iteratively from the top left corner ( $V_{sfx}(0, 0)$ ) to the bottom right corner ( $V_{sfx}(\text{length}(C) + 1, \text{length}(D) + 1)$ ).

$$\begin{aligned} v_{sub}(C, D) &= \max\{v_{sfx}(C', D') : C' \text{ suffix of } C, D' \text{ suffix of } D\} \\ &= \max\{v_{sfx}(C[1:i], D[1:j])\} \\ &= \max\{V_{sfx}(i, j)\} \end{aligned}$$

Using the BLOSUM matrix and  $s(a, -) = s(-, a) = -2$ , the program below (figure 8) calculates the max entry of the matrix and returns it as  $v_{sub}$ .

```
function [vsub] = vsub(S,T,blosum,letters)
    i = length(S);
    j = length(T);
    vsfx = zeros(i+1, j+1);
    for index1 = 2:i+1
        for index2 = 2:j+1
            letterIndex1 = letters==S(index1-1);
            letterIndex2 = letters==T(index2-1);
            a1 = vsfx(index1-1, index2-1)+blosum(letterIndex1, letterIndex2);
            a2 = vsfx(index1-1, index2)-2;
            a3 = vsfx(index1, index2-1)-2;
            vsfx(index1, index2) = max([0 a1 a2 a3]);
        end
    end
    %Return v_sub = max{V_sfx(i,j)}
    vsub = max(vsfx, [], 'all');
end
```

Figure 8: *vsub.m*

Applying this to the keratin protein structures from before gives a  $v_{sub}$  score of 1312.

Reference:

<https://www.maths.cam.ac.uk/undergrad/catam/11/9pt3.pdf>