1 Scoring Function Overview

The scoring function that I have designed builds upon the suggestions in the assignment brief, and to the best of my knowledge has strong biological intuition. The score function is made up of three key parts.

1.1 Convex Gap Scores

It is well-known that single mutation events - such as translocation and duplication - may create alignment gaps of different sizes. Therefore, there is a biological need to treat such gaps as a single entity, as opposed to individually penalising each successive *indel*. Two protein sequences may be relatively similar but differ only at a certain interval, and we don't want to excessively penalise such an alignment. Affine gap scores have been used extensively in industry, and are often the gap score method of choice when partnered with the *BLOSUM* score matrices. An affine gap score combines a constant 'gap-opening' penalty (favouring shorter gaps) with an additional penalty that is linear in the number of further residues in the gap (favouring fewer, larger gaps). Typically, the 'gap-opening' penalty is an order of magnitude larger than the additional penalty ((10, 1) for BLOSUM-62), and thus such a gap-score scheme goes some way to reducing the penalty given to large gaps. However, it has been shown empirically that an affine gap length is too rigid for use in a biological context[1]. Moreover, other studies have shown that the distribution of *indels* typically follows a power law (logarithmic) distribution[2]. Therefore, I have opted to use a convex gap penalty model, in which each additional space in a gap contributes less to the gap weight than the previous space. The model I propose for (internal) gap scores is:

$$P_g \cdot \ln(n+2)$$

where P_g is the gap opening penalty, and n is the length of the gap.

1.2 Trailing, Internal and Terminating Gap Score Differentiation

The second key feature of my scoring function is the implementation of different gap-scoring parameters for trailing, internal, and terminating gaps. It has been shown empirically, through optimization techniques, that the optimal model for matches often has opening and terminal gap-open penalties that are approximately half of the gap-open penalty used for internal gaps[3]. Intuitively, this makes sense - the query sequence could merely be a translation of the database sequence, and should not be excessively penalised. Therefore, the model I propose for trailing and terminating gap sequences is:

$$\frac{P_g}{2} \cdot \ln(n+2)$$

where P_g is the gap opening penalty, and n is the length of the gap. It is worth noting that this feature of the scoring function only applies to the global alignment case, as all (biologically-imitating) score functions assign negative scores to gaps, and so leading and trailing gaps would never be included in a local alignment.

1.3 Codon Match Reward

A codon is a set of three bases (technically three nucelotides) which codes for a certain amino acid. This sequence of contiguous triplets defines a protein's functionality, and the codons hold the key to the translation of genetic information for the synthesis of proteins. Therefore, I believe matches of (multiples of) three contiguous bases should be rewarded with a score that is higher than the summation of the scores of the individual matches. I propose a multiplicative scoring system, where:

$$\textit{Adjusted Score} = C(t) \cdot \textit{Additive Score},$$

where t is the number of codons matched consecutively. This codon scoring scheme could be extend further and the entire alignment scoring could be implemented on the codon level, using the empirical scoring matrices found here[4].

1.4 Further Features

Further features that could be included to more accurately mimic biologically reality include:

- Take into account the position within the current codon. Point mutation frequencies are not evenly distributed over the three positions within a codon, so different scoring matrices could be used for each of the three positions
- Have gap-specific indel scores. There is evidence to suggest that specific residue types are preferred in gap regions[5], and so once a gap has been opened a secondary scoring matrix could be used for the following residues in the gap.
- Score specific mutation events individually. For example, in the case of a duplicated amino acid (a triplet/codon of three bases), the penalty induced for the insertion of three *indels* could be reduced if the three residues are identical to previous triplet of residues: as in the case of AAC**ACG**TCG and AAC**ACGACG**TCG, for example.

[1] Sung, Wing-Kin (2011). *Algorithms in Bioinformatics : A Practical Introduction. CRC Press. pp. 42?47*

[2] http://elbo.gs.washington.edu/courses/GS_559_11_wi/slides/4A-Sequence_comparison-ScoreMatrices.pdf

[3] https://www.ncbi.nlm.nih.gov/pmc/articles/PMC548345/

[4] https://bmcbioinformatics.biomedcentral.com/articles/10.1186/1471-2105-6-134

[5] Wrabl JO, Grishin NV (2004). "Gaps in structurally similar proteins: towards improvement of multiple sequence alignment"

2 Scoring Function Implementation

2.1 Parameters

As I have described an array of different features above, I shall implement only the convex gap score penalty, and the codon match reward function. The parameters required are therefore:

- The gap-opening penalty: $P_g = -10$ (as is often used with some of the BLOSUM matrices)
- The codon match reward function: $C(t) = \begin{cases} 1 + (0.1)t & 0 \leq t \leq 10 \\ 2 & t > 10 \end{cases}$ where $t = \lfloor m/3 \rfloor$, and m is the number of contiguously matched residues. This function increasingly rewards longer contiguous matches of codons.
- The scoring matrix: $a = \left(\begin{smallmatrix} 1 & -1 & -2 \\ -1 & 2 & -4 \\ -2 & -4 & 3 \end{smallmatrix}\right)$, indexed in the usual way for the alphabet $\Sigma = \{A, B, C\}$. In the absence of a real scoring matrix such as PAM or BLOSUM, I shall use the score matrix given in the assignment brief. In reality, the score matrix would be alphabet-specific, and would be found empirically using the probabilistic model discussed in lectures.

2.2 Algorithm

Algorithm 1: Local Sequence Alignment with Convex Gap Penalties and Codon Rewards

```
Input  : A query sequence $Q$, and a database sequence $D$, both formed from the alphabet $\Sigma = \{A, B, C\}$
Output: An array of [Alignment Score, The indices of $Q$, The indices of $D$]
Data:
score_matrix ← $[[1, -1, -2], [-1, 2, -4], [-2, -4, 3]]$
$P_g \leftarrow -4$
$C(t) \leftarrow 1 + (0.1 \cdot t)\ if\ 0 \leq t \leq 10\ else\ 2$
1  $align\_matrix \leftarrow$ zero_array$(m+1, n+1)$; // The dynamic programming matrix of scores
2  $pointers \leftarrow$ zero_array$(m+1, n+1)$
3  for $i,\ l\ in$ enumerate$(D)$ do
4      for $j,\ k\ in$ enumerate$(Q)$ do
5          $left\_gap, a, b \leftarrow$ backtrack$(i, j, left)$;
6          $up\_gap, c, d \leftarrow$ backtrack$(i, j, up)$;
7          $num\_matched \leftarrow$ backtrack$(i, j, diag)$;
8          $left\_score \leftarrow align\_matrix[a, b] + (P_g \cdot \ln(left\_gap + 2))$;
9          $up\_score \leftarrow align\_matrix[c, d] + (P_g \cdot \ln(up\_gap + 2))$;
10         $diag\_score \leftarrow align\_matrix[i, j] + (score\_matrix[l, k] \cdot \mathrm{C}(\lfloor \frac{num\_matched}{3} \rfloor)$;
11         $scores \leftarrow$ array$(0, left\_score, diag\_score, up\_score)$;
12         $align\_matrx[i+1, j+1] \leftarrow$ max$(scores)$;
13         $pointers[i+1, j+1] \leftarrow$ argmax$(scores)$; // 0 - New alignment, 1 - Left, 2 - Diagonal, 3 - Up
14     end
15 end
16 return [max($align\_matrix$)), generate_indices($pointers,$ argmax($scores$))]
```

The implementation of my algorithm follows the same structure as the standard quadratic-space dynamic programming algorithm implemented in the first part of the assignment. That is, it finds optimal alignments of shorter subsequences of the two input sequences, and 'recursively' builds up a larger alignment from the previous smaller alignments. However, there are some key changes. To facilitate the convex gap penalties, the algorithm must know when the current gap began if the current path being extended did indeed insert a gap in the previous location. The *backtrack* function takes care of this, by recursively following the pointers back in the direction specified, starting from coordinates specified, until the path being considered takes a step in a direction different to the one specified. For example, calling *backtrack(i, j, left)* will cause the algorithm to follow the pointers back from the (i, j) location in the matrix, until the next direction of movement is not left. As a movement left in the matrix corresponds to inserting a space into the sequence on the 'left' of the matrix, doing so allows us to find the beginning of the current gap, and thus its length. When *left* or *right* is passed to the *backtrack* function, the length and starting coordinates of the gap are returned. We then apply a logarithmically-scaled penalty to the gap, which means that successively longer gaps incur a decreasing additional penalty. The scaling factor of + 2 is included so that opening a gap (which corresponds to a current gap size $n = 0$) is not penalised infinitely (in the case of $\ln(0+0)$), but does indeed incur a penalty (which wouldn't be the case with $\ln(0 + 1) = \ln(1) = 0$). Finally, we return the maximum