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:

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

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:

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

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:

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

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: $a = \begin{pmatrix} 1 & -1 & -2 \\ -1 & 2 & -4 \\ -2 & -4 & 3 \end{pmatrix}$, 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 $\leftarrow [[1, -1, -2], [-1, 2, -4], [-2, -4, 3]]$
$P_g \leftarrow -4$
$C(t) \leftarrow 1 + (0.1 \cdot t)\ if\ 0 \leq t \leq 10\ else\ 2$

```
align_matrix ← zero_array(m + 1, n + 1);
pointers ← zero_array(m + 1, n + 1);
for i, l in enumerate(D) do
    for j, k in enumerate(Q) do
        left_gap, a, b ← backtrack(i, j, left);
        up_gap, c, d ← backtrack(i, j, up);
        diag_gap ← backtrack(i, j, diag);
        left_score ← align_matrix[a, b] + (P_g · ln(left_gap + 2));
        up_score ← align_matrix[c, d] + (P_g · ln(up_gap + 2));
        diag_score ← align_matrix[i, j] + (score_matrix[l, k] · C(⌊diag_gap / 3⌋));
        scores ← array(0, left_score, diag_score, up_score);
        align_matrx[i + 1, j + 1] ← max(scores);
        pointers[i + 1, j + 1] ← argmax(scores);
    end
end
while While condition do
    instructions ] ← test;
    if condition then
        instructions1;
    else
        instructions3;
    end
end
```

A few small prerequisites need to be installed before running my program. These are: Python 3, GNU Multiple Precision Arithmetic Library (GMP), and the SoPlex solver. I shall include the GMP and SoPlex archived files (for installation on a Unix-based system) in my final submission. Alternatively, follow the blue links to go to their respective download pages. To install these dependencies, please follow the instructions below:

- **GNU Multiple Precision Arithmetic Library**
  Please change directory to where the *gmp-6.1.2.tar.lz* file is located and run the following commands:

```
lzip -d gmp-6.1.2.tar.lz
tar -xf gmp-6.1.2.tar
cd gmp-6.1.2
./configure
make
make check
make install
```

- **SoPlex Solver**
  Please change directory to where the *soplex-4.0.1.tgz* file is located and run the following commands:

```
tar -xf soplex-4.0.1.tgz
cd soplex-4.0.1
mkdir build
cd build
cmake <path/to/SoPlex> -DGMP=true
make
make test
make install
```

- **Pyomo**
  The easiest way to install Pyomo is to use Python's built-in package manager, *pip*. Once *pip* has been installed, please run the following command:

```
pip install pyomo
```

Should you run into any issues installing any of these dependencies, please consult the relevant INSTALL files and documentation.

3 Usage

3.1 Directory Structure

Once the dependencies have been installed, the directory structure should look something like Figure 1 (varying slightly if you installed the dependencies in a different location):

Note: The program will **not** work unless the four folders (*graphs*, *models*, *settings*, *solutions*) are present. This is what is contained in each folder:

- **/graphs**: This folder contains the input graph files. Graph files should be saved with a *.txt* extension, and follow the following format:

```
[V] [E]
[v11] [v12]
[v21] [v22]
 ...
[vE1] [vE2]
```

  where **V** is the number of vertices, **E** is the number of edges, and each [**vi1**] [**vi2**] pair (for i $\in$ {1, 2, ... , E} represents the edge between vertex [**vi1**] and [**vi2**]. For example, the complete four-vertex graph would be represented as follows:

```
4 6
0 1
0 2
0 3
1 2
1 3
2 3
```

  Note: Vertex numbering *must* start at 0, not 1, for the program to work.

- **/models**: This folder contains the *.mps* linear program modelling files generated by my program, which are then solved by the SoPlex solver.

- **/settings**: This folder contains the settings files for the SoPlex solver.

- **/solutions**: This folder contains the detailed solutions outputted by the SoPlex solver. Each time the program is executed, it will output two files - one for the Fractional Clique Cover Number, and one for the Shannon Entropy.

3.2 Running

To run the program, please run the following command (from the base directory):

```
python Assignment.py graphs/[graphName.txt]
```

The program will output the Fractional Clique Cover Number, the Shannon Entropy, and the solutions to both of the linear programs concisely, and in rational format. As mentioned above, more detailed output for both linear programs is saved in the /solutions folder.

4 Linear Program Reformulation

4.1 Fractional Clique Cover Number

- In the specification, variables are introduced for all the subsets S of V, the set of all vertices. However, as the only non-zero variables are those corresponding to the cliques of V, I introduced variables only for the cliques. To find the cliques of V efficiently, I modified the well-known *Bron Kerbosch* algorithm, so that it recursively finds *all* the cliques of V, not only the maximal ones.

- Whilst searching for how to further simplify the linear program, I found an article by Peter Cameron[1] which states that *"It can be shown that the same minimum value [the Fractional Clique Cover Number] is obtained if we restrict to regular fractional clique covers."*. A regular fractional clique cover is one where, for every vertex, the sum of the values given to the cliques that contain this vertex **equals** one, rather than being bounded above by 1. The proof of this claim can be found in the paper referenced in the article[2], and so I implemented this simplification in my program.

4.2 Shannon Entropy

For the Shannon Entropy linear program, variables are required for all of the subsets of the set of vertices V. However, many of the constraints 'overlap' and so are redundant.

- The first constraint that I simplified was the:

$$x_T - x_S \geq 0 \quad \forall S \subseteq T \subseteq V$$

  constraint. For very subset T of V, we do not need to create a constraint for every subset S of T; it is sufficient to only create constraints for subsets S that are of size one less than T. This is because if $R \subseteq S \subseteq T$, then $x_R \leq x_S$ and $x_S \leq x_T \implies x_R \leq x_T$. This is more easily shown diagramatically:

  For a graph of N vertices, the total number of constraints without the reformulation would be $\sum_{n=2}^{N} \binom{N}{n}(2^n - 1)$ (where constraints involving the empty set have been removed as the variable corresponding to the empty set is 0), whereas the the number of constraints with this reformulation is $\sum_{n=2}^{N} \binom{N}{n} n$. For the N = 8 case, this is a reduction of $6297 - 1016 = 5281$ constraints.

- To increase the efficiency of the constraint generation, I calculated the above constraint and the $X_{N(v)\cup\{v\}} - x_{N(v)} = 0$ constraint in one traversal of the subsets pairs, as the neighbourhood constraint also involves two sets differing in size by only one variable. I don't think it is possible to further simplify the neighbourhood constraint, and as there is only one constraint per vertex, the number of these constraints is negligible with respect to the total number of constraints.

- The next constraint that I simplified significantly was the:

$$x_S + x_T - x_{S\cup T} - x_{S\cap T} \geq 0 \quad \forall S, T \subseteq V$$

  constraint. Instead of creating a constraint for every pair of vertices S and T, of which there would be $2^N \times 2^N = 2^{2N}$, constraints are only required for pairs of vertices where $S \not\subseteq T$ and $T \not\subseteq S$. If $S \subseteq T$, then:

$$x_S + x_T - x_{S\cup T} - x_{S\cap T} = x_S + x_T - x_T - x_S = 0$$

  and so the constraint becomes trivial. The same argument holds for $T \subseteq S$ by simply switching the variable labels.

- The final constraint I attempted to simplify was the:

$$x_{\{v\}} \leq 1 \quad \forall v \in V$$

  constraint. After lots of experimentation, I believe that the inequality in this constraint can be replaced with an equality. This intuition is backed up by the proof that this modification holds in the Fractional Clique Cover Number linear program. Furthermore, when testing my program with the constraint as an inequality, SoPlex reported that there were a small number of redundant constraints, which I believe to relate to this constraint. However, as I could not come up with a formal proof to back up this intuition, I decided to leave the constraint as an inequality in my final implementation.

5 Examples

In Figure 3, the outputs for the complete eight vertex graph and Graph 1 are shown. As expected, the Fractional Clique Cover Number of the complete graph is 1, the Shannon Entropy of the complete graph is $8 - 1 = 7$, the Fractional Clique Cover Number of Graph 1 is $10/3$, and finally the Shannon Entropy of Graph 1 is $11/3$. Please note that multiple other example graphs are included in the /graphs folder, including all 6 graphs from the paper[3].

6 Additional Information

- If you would like the program to print out the linear program (the objective function and the constraints) before it is solved, please uncomment out line 435 of *Assignment.py*.
- I inverted all of the greater than constraints so that the model only consists of equality and less than constraints, in order to simplify the implementation.
- The vertex numbering in the input files must begin at vertex **0**, and continue sequentially.
- In my initial implementation, I used the linear program solver included in the SciPy package, and then used the Fraction module to convert the floating point output to a rational format. However, this seemed unsatisfactory and susceptible to floating-point arithmetic nuances, and so I reimplemented my code using Pyomo and SoPlex. My program first generates all the variables and constraints, which are then passed to Pyomo. Pyomo builds the LP model and outputs it in the standard *.mps* format, and then my program spawns a new process and executes the SoPlex program, which solves the model. My program then reads the exact solution outputted by SoPlex, parses the rational numbers as strings, creates exact fractions from these strings using the Fraction module, and then finds the exact optimal value using the exact optimal solution and the objective function.

References

[1] Peter Cameron: *Guessing Numbers of Graphs*,
https://cameroncounts.wordpress.com/2016/03/20/guessing-numbers-of-graphs/

[2] Peter Cameron, Anh N. Dang, Søren Riis: *Guessing Games on Triangle-free Graphs*,
The Electronic Journal of Combinatorics

[3] Maximilien Gadouleau: *On the possible values of the entropy of undirected graphs*,
Journal of Graph Theory