DNA Sequence Alignment Non-Gap Penalties and Gains

Justin Bass

**Abstract**

Global sequence alignment is important for best-quality alignments between two DNA sequences. There has been much work done on the effects of varying types of sequential-gap penalty functions, but never on the effects of similar sequential-match and sequential-mismatch functions besides for the standard linear ones in common use. In this paper, I investigate the effects of such scoring functions and their effects on alignment accuracy. I also explain the observed effects by looking at the effects on gap frequency and length in the corresponding alignments. The results of simulation indicate that performing these modifications on sequential-matches has a worse effect on accuracy than linear, but sequential-mismatch functions are markedly improved by the addition of a simple opening-penalty. However, running this simulation is almost prohibitively expensive compared to just simulating gap penalty functions, so it is probably not worth the effort for such a small increase in efficiency.

**Background**

Sequence alignment is an important component of genome comparison or assembly. When the quality of alignment is important, a global sequence alignment is often used, ensuring that the canonical scoring function is globally maximized. One of the most commonly used algorithms is the Needleman-Wunsch algorithm, which uses a dynamic programming approach to reduce the time complexity to O(N²) [1]. The scoring consists of three components: matches, mismatches, and gaps. The most commonly used functions for each of the three are linear for matches, linear for mismatches, and affine for gaps. There have been several papers discussing the use of more complex gap penalty functions such as logarithmic or affine logarithmic [2]. These have been motivated by empirical results, although turned out to be inappropriate for alignment [3]. Despite this focus on gap functions, few papers acknowledge the use of non-linear match gains and mismatch penalties (here referred to as non-linear match and mismatch functions, NLM). This is due in part to the large complexity of testing different coefficients for these functions, which is a cubic increase in simulation time complexity compared to just altering the gap penalty function. Still, it is possible to perform the Needleman-Wunsch algorithm with arbitrary scoring functions without an increase in time complexity, as is implied by previous research [4,5].

In this paper, I hypothesized that NLM functions would have a significant positive effect on scoring outcome. A small subset of all possible NLM functions were explored with simple coefficients. The results indicate that an opening penalty for sequential mismatches is beneficial to score by discouraging mismatches with no effect on gaps, but complex match functions decrease the accuracy by encouraging more gaps to form.

**The Needleman-Wunsch Alignment Algorithm**

The Needleman-Wunsch algorithm takes as input 2 strings (string1 and string2) and outputs an aligned set of strings (align1, align2) along with a score. The algorithm uses a dynamic programming approach seen in Figure 1. The two strings are laid orthogonally to create a grid, which is used to keep track of the best alignment at each point.

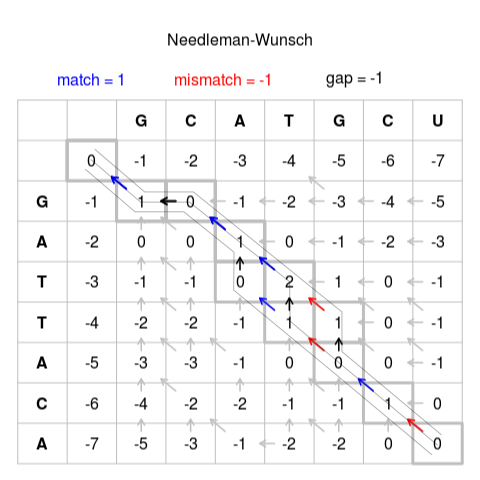


Figure 1: Aligning 'GCATGCU' and 'GATTACA'

For each cell, assuming the general case, the best score is calculated as:

score[y,x] = max(score[y-1,x] - gap(L), score[y-1,x-1] - mismatch(L), score[y,x-1] - gap(L))

This assumes the characters corresponding to the current cell (str1[x] and str2[y]) are not the same. If the two characters do match, which is the case in cell (1,1) where the first characters of str1 'GCATGCU' and str2 'GATTACA' match, then the score is calculated as:

score[y,x] = max(score[y-1,x] - gap(L), score[y-1,x-1] + match(L), score[y,x-1] - gap(L))

The meaning of this equation is as follows: a step to the right indicates the creation of a gap in str1, which necessitates a reduction of the previous score in that cell (to the left) by the gap function. A step down indicates the creation of a gap in str2, which necessitates a reduction of the previous score in that cell (above) by the gap function. A step diagonally indicated the alignment of the two characters, matching or mismatching, and a gain or penalty is taken from the previous score in that cell (diagonally).

Although in this example the scores are all linear with respect to length, more complicated functions can be applied, and I explain the alteration in the methods section.

One this has been performed, the bottom-right most cell corresponds to the complete alignment of both two strings. We back-track through the score matrix, following the largest score to either the top, left, or top-left, and keeping track of these directions along the way. In my implementation of the algorithm, if two directions can be traversed, say top-left and top, I always prefer the top-left, to potentially decrease the total number of gaps. Generally, multiple alignments can be returned from this algorithm, but my scoring method requires a single alignment. As the algorithm ensures all returned alignments will have the same score, this is inconsequential, but since I also look at the number of gaps separately, it might be useful to prefer alignments first.

Back-tracking up corresponds to a gap in str2, to the left a gap in str1, and top-left an alignment of the two characters. When the cell (0,0) is reached, the entirety of the two strings has been aligned. The total score can be counted as well during back-tracking by using the match, mismatch, and gap functions.

**Methods**

I altered the Needleman-Wunsch algorithm by adding 3 matrices to track the length of each set of sequential indels (H for and V), matches, and mismatches (Dm, Dmm). Iterating through the score matrix, each time a match, mismatch, indel in string1, or indel in string 2 is opened, the respective matrices Dm, Dmm, H, and V are incremented by 1. For each subsequent one, the count is continually incremented by 1. When a different one is performed, the count resets to 0. This way, the length of each of the three is tracked. To calculate the match, mismatch, or indel scoring with an arbitrary function F, the score is essentially incremented by:

F(len) - F(len - 1) : len > 1

F(len) : len = 1

0 : len = 0

Thus, at each step the score will be accurate to the current length of each type. The algorithm proceeds normally, backtracking through the scoring matrix to find the most optimal pathway of alignment, then returning it.

The actual 11 scoring functions used were: None [0], Linear [L], Affine [1 + L], Affine-log [L + ln(L+1)], Affine-log2 [1 + L + ln(L+1)], Log [ln(L+1)], Log2 [1+ln(L+1)], Quad [L^2], Quad2 [1+L^2], SubQuad [L^1.5], and SubQuad2 [1+L^1.5].

For each trial of my simulation, string1 was randomly generated from the alphabet [A,G,T,C] with length 100. Then, string2 was generated by performing indels, matches, mismatches, and CNVs, resulting in an average length of 112. Indels in particular were biologically accurate [3]. The number of actual matches, mismatches and indels is stored (a CNV counted as indels). Then the algorithm is run for each function on the same strings, and the measured matches, mismatches and indels are calculated from the alignments. A score is calculated as follows:

score = matches / (matches + mismatches + indels)

The measured and actual scores are compared, and the final score is the percentage difference between the two, always less than 1. This procedure was run for 100 trials, and the mean and standard deviations of the scores are generated. Running 100 trials appears to give a reasonably consistent result, within a few percent between runs. Little additional accuracy is imparted by running more than 100 trials, so the cost in time does not justify the means.

**Results**

Data was collected by considering each function for matches, mismatches, and indels independently of the other two functions. The resulting number represents the average performance of each function for a broad but consistent number of other scoring functions.

|  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- |
| Scores | Match (mean) | Match (std) | Mismatch (mean) | Mismatch (std) | Gap (mean) | Gap (std) |
| None | 0.670 | 0.140 | 0.611 | 0.105 | 0.627 | 0.078 |
| Linear | **0.822** | 0.099 | 0.739 | 0.102 | 0.768 | 0.087 |
| Affine | 0.737 | 0.100 | **0.775** | 0.097 | **0.802** | 0.095 |
| AffineLog | 0.758 | 0.099 | 0.757 | 0.100 | **0.783** | 0.097 |
| AffineLog2 | 0.615 | 0.088 | **0.779** | 0.096 | **0.788** | 0.099 |
| Log | 0.729 | 0.100 | 0.688 | 0.112 | 0.671 | 0.103 |
| Log2 | 0.565 | 0.088 | 0.747 | 0.106 | 0.742 | 0.110 |
| Quad | **0.812** | 0.100 | 0.743 | 0.101 | 0.722 | 0.107 |
| Quad2 | **0.807** | 0.100 | **0.777** | 0.095 | 0.733 | 0.121 |
| SubQuad | **0.813** | 0.100 | 0.739 | 0.102 | 0.737 | 0.102 |
| SubQuad2 | **0.802** | 0.100 | **0.775** | 0.096 | 0.756 | 0.114 |

Table 1: Resulting Scores for the Three Function Types

Table 1 summarizes the results. In red are the highest scores per column, and bolded are results within 2% of the max, likely to be maximum as well due to the variance between runs and limited number of trials. As a sanity check, using the None function returns the worst results in general. Also, gap penalties are most accurately Affine-log or Affine with Linear being the next best choice, and Log and Quad being the worst, which has been found by other studies [2].

The best match function is Linear, with Quad and SubQuad performing almost as well. For mismatches, Affine-Log2, Quad2, SubQuad2, and Affine perform the best, significantly better than linear.

|  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- |
| Gap Size | Match (gap num) | | Match (gap len) | Mismatch (gap num) | Mismatch (gap len) | Gap  (gap num) | Gap  (gap len) |
| None | 34.232 | 8.163 | | 31.886 | 3.584 | 52.181 | 3.240 |
| Linear | 24.651 | 2.484 | | 34.522 | 2.627 | 34.195 | 2.202 |
| Affine | 37.053 | 1.912 | | 33.932 | 2.523 | 26.677 | 2.683 |
| AffineLog | 34.315 | 2.032 | | 34.913 | 2.576 | 28.426 | 2.450 |
| AffineLog2 | 54.411 | 1.576 | | 34.173 | 2.508 | 25.554 | 2.956 |
| Log | 32.575 | 2.131 | | 35.420 | 2.919 | 48.685 | 3.304 |
| Log2 | 53.705 | 1.583 | | 34.525 | 2.634 | 37.949 | 4.665 |
| Quad | 26.205 | 2.527 | | 34.966 | 2.536 | 34.340 | 1.712 |
| Quad2 | 27.246 | 2.336 | | 34.254 | 2.485 | 27.494 | 2.146 |
| SubQuad | 25.854 | 2.503 | | 35.033 | 2.554 | 34.778 | 1.859 |
| SubQuad2 | 27.678 | 2.210 | | 34.302 | 2.510 | 27.646 | 2.241 |

Table 2: Resulting Gap Sizes for the Three Function Types

Table 2 summarizes the effects of the functions of gap length in an attempt to explain the underlying reasons behind the decreased accuracy for complex match functions. As a sanity check, opening penalties in the Gap function's gap num and gap len columns (e.g. the difference between Log and Log2) cause the number of gaps in the output alignment to decrease, and the length of each gap to increase, as expected.

**Discussion**

It can be seen that opening penalties in match functions have the opposite effect of increasing the number of gaps and decreasing the number of each gap's lengths. This is because non-linear match functions tend to group matches together, in effect stealing away from the effects of grouping indels together.

On the other hand, opening penalties in the mismatch functions have no significant effect on gap number or length, but discourage mismatches from forming and thus increase the score. However, it should be noted that the specific function seems to be almost irrelevant - any use of an opening penalty for mismatches is beneficial to scoring.

**Future Work**

The data's accuracy is limited by the number of trials, the length of string1, and the number of functions that could be tested. On my computer's Intel® Core™ i7 @ 2.5GHz processor, the estimated computation time can be calculated by:

Time = 3.4 µs \* trials \* length² \* functions³

Thus my simulation with 100 trials, 100 length, and 11 functions took just a few hours to run. However, ideally 1000 functions could be surveyed to see if there are any functions that scored better, which might then justify the use of NLM functions. Also, strings of length 1000 would be more appropriate for real-life alignment scenarios. This would cause the algorithm to take 10,000 years to run, but as it is highly parallelizable (each trial and function is independent), sufficient cloud-computing or super-computing would allow this. Thus future work might include implementing parallelization to increase CPU utilization, and a more capable array of processors or GPUs.

References

1. Needleman, SB & Wunsch, CD. (1970) A general method applicable to the search for similarities in the amino acid sequence of two proteins. J. Mol. Biol., 48:443-453.

2. Cartwright, Reed A. (2006) Logarithmic gap costs decrease alignment accuracy. BMC Bioinformatics, 7:527.

3. Gu X, Li WH: The size distribution of insertions and deletions in human and rodent pseudogenes suggests the logarithmic gap penalty for sequence alignment. J Mol Evol 1995, 40:464-473.

4. Mott, Richard. (1999) Local sequence alignments with monotonic gap penalties. Bioinformatics, 15(6):455-462.

5. Altscul, SF. (1998) Generalize affine gap costs for protein sequence alignment. Proteins, 32:88-96.

6. Jennifer Kavran. The Role of Gap Penalties in Searching for Distant Homologues. MBB 752A2 Final Project.