Computational Biology
Lecture 6: Affine gap penalty function, multiple sequence alignment
Saad Mneimneh

We saw earlier how we can use a concave gap penalty function γ, i.e. one that satisfies $\gamma(x+1)-\gamma(x) \leq \gamma(x)-\gamma(x-1)$, as a more realistic model for penalizing gaps. This was at the expense of increasing the time complexity of our dynamic programming algorithm. With a concave gap penalty function, the new DP algorithm requires $O(mn^2 + nm^2)$ time, as compared to the $O(mn)$ time bound of previous algorithms. We will seek a solution that will bring us back to our old time bound of $O(mn)$ while still providing a good model for gap penalty.

Affine gap penalty function

The objective is to come up with a realistic gap penalty function that will not require the dynamic programming algorithm to spend more than $O(mn)$ time updating the table. Let us look then at the origin of the problem and why the general gap penalty function fails to achieve that: The general penalty function can possible have a different penalty for each additional gap. The algorithm has to check for all possible gap lengths. At a given position $A(i, j)$ this is $i + j$ possibilities. This is what makes the algorithm spend cubic time for updating all entries.

We will approximate γ by a set of linear functions. For simplicity assume that we will approximate it using two functions as illustrated below (this could be generalized to more than two).

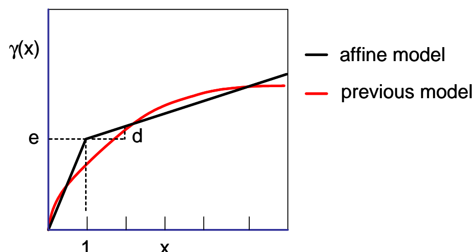

Figure 1: Affine gap penalty function

The figure above suggests that we penalize the first gap differently, possibly more than the rest, then each subsequent gap will be penalized linerarly. The gap penalty function will be

$$\gamma(x) = e + (x-1)d, \quad x \geq 1$$

This is called the affine gap penalty model. If $e >> d$, which is usually the case, then this is still a concave function and therefore a good approximation of the realistic setting. However, for the rest of this document, we do not impose any constraints on d and e (concavity will not be a requirement).

What have we gained so far? Now the gap is either the opening of a larger gap or not. We only distinguish between these two cases. Therefore, we have only two possibilities to check for. If we have an opening of the gap, we penalize it by e; otherwise, we penalize as before by d. But now we have to be able to detect the opening of a gap.

To be able to detect the opening of a gap, we will explicitly distinguish between three kinds of alignments: alignments that end with no gaps, alignments that end with a gap aligned with x, and alignments that end with a gap aligned with y. For this purpose, we are going to define three types of blocks in an alignment:

- A: block containing one character in x aligned to one character in y
- B: block containing a *maximal* number of contiguous gaps aligned with x, thus a type B block cannot follow a type B block.
- C: block containing a *maximal* number of contiguous gaps aligned with y, thus a type C block cannot follow a type C block.

Therefore, an alignment can be divided into blocks (instead of columns as before) of three types A, B, and C. The following is an example:

```
A|A|C|---|A|ATTCCG|A|C|T|AC
A|C|T|ACC|T|------|C|G|C|--
```

An important feature of the division above is that the score of the alignment becomes additive across blocks boundaries (this is true regardless of the gap penalty function). Therefore, we can make our dynamic programming table maintain three matrices for the three kinds of alignments mentioned above: alignments that end with a block of type A, alignments that end with a block of type B, and alignments that end with a block of type C. Each entry in one of these three matrices will now depend on previous entries in other matrices. For instance, the optimal score for an alignment of $x_1...x_i$ and $y_1...y_j$ that ends with a gap aligned with x (defined as $B(i, j)$) is obtained by either making the gap an opening gap, thus $A(i-1, j) - e$ or $C(i-1, j) - e$, or making the gap a continuation of a previously opened gap, thus $B(i-1, j) - d$. Obviously, the maximum of those three is the optimal score for $B(i, j)$.

More formally:

- $A(i, j)$: optimal score for an alignment of $x_1...x_i$ and $y_1...y_j$ that ends with no gaps
- $B(i, j)$: optimal score for an alignment of $x_1...x_i$ and $y_1...y_j$ that ends with a maximal gap aligned with x
- $C(i, j)$: optimal score for an alignment of $x_1...x_i$ and $y_1...y_j$ that ends with a maximal gap aligned with y

Then our update rules become:

$$A(i,j) = \max \begin{cases} A(i-1,j-1) + s(i,j) \\ B(i-1,j-1) + s(i,j) \\ C(i-1,j-1) + s(i,j) \end{cases}$$

$$B(i,j) = \max \begin{cases} A(i-1,j) - e & different\ block,\ opening\ gap \\ B(i-1,j) - d & same\ block,\ continuing\ gap \\ C(i-1,j) - e & different\ block,\ opening\ gap \end{cases}$$

$$C(i,j) = \max \begin{cases} A(i,j-1) - e & different\ block,\ opening\ gap \\ B(i,j-1) - e & different\ block,\ opening\ gap \\ C(i,j-1) - d & same\ block,\ continuing\ gap \end{cases}$$

Special care has to be made for the initial conditions of the three matrices. $A(0, 0) = 0$. $A(i, 0) = -\infty$ for $i = 1...m$ and $A(0, j) = -\infty$ for $j = 1...n$. The reason for this is that $A(i, 0)$ and $A(0, j)$ do not correspond to alignments that end with type A block, and hence we have to make sure they do not affect the computattion. Similarly, $B(0, 0) = 0$, $B(i, 0) = -e - (i-1)d$, and $B(0, j) = -\infty$. Finally, $C(0, 0) = 0$, $C(i, 0) = -\infty$, and $C(0, j) = -e - (j-1)d$.

The trace back pointers are kept as before, except that now they can jump from one matrix to another and we start from $\max(A(m, n), B(m, n), C(m, n))$.

We can simplify the above algorithm a little bit. Note that the following two types of alignments cannot be optimal if $s < d + e$.

```
...-x...
...y-...

...x-...
...-y...
```

This is because if $s < d + e$, then the penalty for a mismatch is less then the penalty of two gaps and the two above alignments are not optimal. In other words, a type C block cannot follow a type B block and viceversa. Therefore, if $s \leq d + e$ we can disregard updating B using C or C using B. As a result, if $s \leq d + e$, the third line in the update for $B(i, j)$ can be ignored, and the second line in the updated for $C(i, j)$ can be ignored.

Multiple sequence alignment

The objective is now to generalize what we have seen so far to multiple sequences. Therefore, we need to align k sequences in the optimal way. But what is optimal now? Let us assume for the rest of this document that the score is additive, and hence, the score of an alignment is the sum of scores of each column in the alignment. We have now k characters in each column, as opposed to just 2. Our scoring function will have to takes k arguments now. This will be a complicated function to build. We would like to still use the scoring function for two characters to score the whole column.

One famous scoring shceme is the sum of pairs score score, $SP-score$, which consists of scoring every pair of sequences separately and adding up the scores. Given two sequences s_i and s_j (note that the subscript now refers to the sequence

but not a character in the sequence), define the induced alignment of s_i and s_j as the alignment obtained by isolating both sequences from the multiple alignment and ignoring all columns with all gaps. Let $score_{ij}$ be the score of the induced alignment of s_i and s_j, the the total score of the multiple alignment is $\sum_{i<j} score_{ij}$.

Alternatively, one could look at each column of the multiple alignment and compute the score of the column separately (assuming additivity) as the sum of scores of every pair of characters in that column. In this case, we need to define $s(-,-)$, the score of a gap aligned with a gap. This scenario is now possible since we have k sequences and hence we can have up to $k-1$ gaps in a column. We therefore define $s(-,-)=0$ to be consistent with the score of induced alignments. In other words, if $s(-,-)=0$, then

$$\sum_l SP-score_l = \sum_{i<j} score_{ij}$$

where $SP-score_l$ is the sum of pairs score of column l.

Here's an example: If we have a column $[I,-,I,V]$, then the score will be

$$SP-score(I,-,I,V) = s(I,-)+s(I,I)+s(I,V)+s(-,I)+s(-,V)+s(I,V)$$

Note that the $SP-score$ is independent of the order of characters in a column (which is nice).

An important thing to realize is that the induced alignment between a pair of sequences to is not necessarily an optimal one. The following example shows how to compute the $SP-score$ and illustrates a case where the induced alignment is not optimal.

```
s1= ATG, s2 = ATG, s3 = A, s4 = T

for the multiple alignment:

ATG
ATG
A--
-T-

The score is:

SP-score = SP-score(A,A,A,-) + SP-score(T,T,-,T) + SP-score(G,G,-,-)

Alternatively, using induced alignments:

SP-Score = score(ATG, ATG) + score(ATG, A--) + score(ATG, -T-)
         + score(ATG, A--) + score(ATG, -T-) + score(A-, -T)
         = 3 -3 -3 -3 -3 -4 = -13

Note that the induced alignment for s3 and s4 is:

A-
-T

which is not the optimal alignment for these two sequences.
```

Multiple sequence alignment can be solved using the same dynamic programming formulation as before.

We have now k sequences of length n_i each, $i=1...k$. Therefore we need a k dimentional array A of length n_i+1 in each dimension (recall for the case of two sequences of length m and n we need an array of size $(m+1)\times(n+1)$). Therefore, A requires now $O(n^k)$ space.

$A(i_1,...,i_k)$ holds the score of the optimal alignment of $s_{1_1}...s_{1_{i_1}}$, ..., $s_{k_1}...s_{k_{i_k}}$.

The problem now is that $A(i,j)$ depends on 2^k-1 other entries (note that $k=2 \Rightarrow A(i,j)$ depends on 3 entries, namely $A(i-1,j-1)$, $A(i,j-1)$, and $A(i-1,j)$). To see this, let us look at the simple case of two sequences. For two sequences x and y, $A(i,j)$ can end in 3 different ways, namely x_i aligned with x_j, x_i aligned with a gap, or y_j aligned with a gap. In other words, all possible ways of having gaps at the end, including no gaps at all, and excluding the possibility of having gaps only. Similarly, with k sequences, we can have any combination of gaps at the end, excluding the possibility of gaps only. These are 2^k-1 possibilities. To think of it in another way, consider a vector of length k

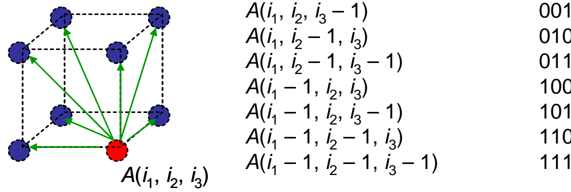

Figure 2: $2^k - 1$ dependencies

of zeros and ones where zero denotes a gap, then we can have all possible vectors except for 000....0. Again, these are $2^k - 1$ vectors. The figure below illustrates for $k = 3$ the $2^k - 1 = 7$ dependencies of $A(i_1, i_2, i_3)$.

Since the $SP - score$ of a column requires $O(k^2)$ time to computes (looking at all pairs), the running time of the dynamic programming algorithm will be $O(2^k k^2 n^k)$.

The exponential time bound for the dynamic programming algorithm is not a surprise. The problem of multiple sequence alignment with $SP - score$ is known to be NP-complete. Therefore, no polynomial time algorithm is known for the problem. For this reason, we will look at heuristic algorithms for the problem. One of the famous heuristics is Star alignment, which is a special case of a more general kind of alignments knows as tree alignments. We will describe what a tree alignment is next.

Tree alignment: Given a tree with k nodes representing k sequences s_1,...,s_k, a multiple alignment of the k sequences *consistent* with the tree is such that the induced alignment between s_i and s_j is optimal if there is an edge (s_i, s_j).

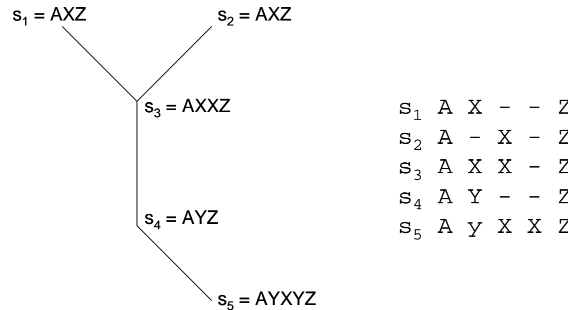

Figure 3: Tree alignment

In the example above, the induced alignment of s_1 and s_3 is optimal. The induced alignment of s_2 and s_4, for instance, is not.

Given a tree, is it always possible to produce an alignment consistent with the tree? The answer is yes. Here's the algorithm to produce such an alignment.

- (1) Pick s_i and s_j such that (s_i, s_j) is an edge in the tree and align them optimally. Let $S = \{s_i, s_j\}$, the set of aligned sequences.
- (2) Pick $s_k \notin S$ and $s_l \in S$ such that (s_k, s_l) is an edge in the tree and align them optimally.
- (3) **Once a gap always a gap**: For each gap added to s_l in this alignment, add a corresponding gap to all sequences in S. For each gap already in s_l, add a corresponding gap in s_k (if needed).
- (4) $S = S \cup \{s_k\}$
- (5) Repeat from (2) until all sequences are in S.

The proof that the algorithm above works is by induction on the number of sequences in S. The base case is when we have two sequences that are optimally aligned and corresponding to some edge (s_i, s_j). The inductive step is when a sequence $s_k \notin S$ is added and aligned optimally with $s_l \in S$ for edge (s_k, s_l) (for a given s_k, there is only one such s_l, since the edges belong to a tree). Using the **once a gap always a gap** strategy, the score of induced alignments for sequences aleardy in S in unchanged (because $s(-, -) = 0$). For the same reason, the score of the new optimal alignment of s_k and s_l is unchanged. Therefore, we obtain a new set $S = S \cup \{s_k\}$ with one additional sequence and a multiple alignment that is still consistent with the tree.

What is the running time of the tree alignment algorithm? Well, we perform $k-1$ optimal alignments (the number of edges in the tree) each taking $O(n^2)$ time. Moreover, every time we add a new sequence to the alignment, we have to update the gaps for all sequences in the alignment (this is $O(k)$ sequences), using **once a gap always a gap** strategy. This gap update will therefore take $O(kl)$ time for each added sequence, where l is the length of the multiple alignment. As a result, the total running time is $O(kn^2 + k^2l)$.

The following figure illustrates the steps of the algorithm on the example presented earlier.

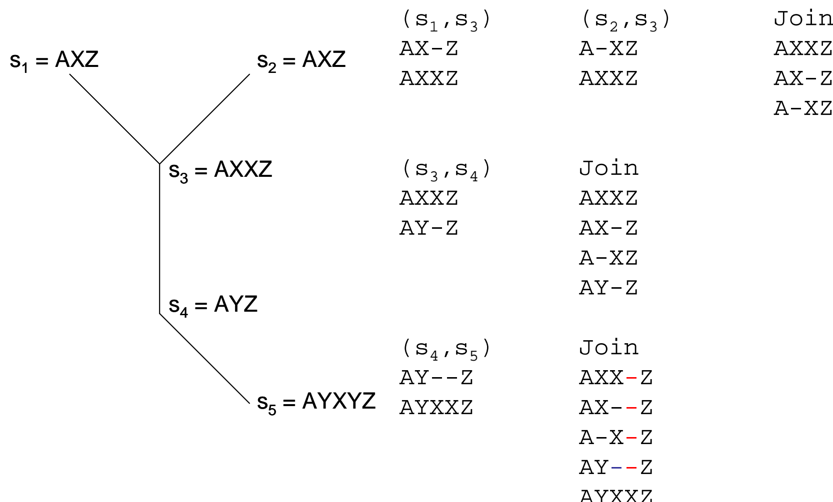

Figure 4: Performing the tree alignment algorithm

Now back to Star alignment. The Star alignment is just a special case when the tree is a star. One sequence will be the center of the star and every other sequence is connceted to the center by an edge.

The question of course is, given the sequences s_1, ..., s_k, which one to choose to be the center of the star? The only guarantee of Star alignment is to produce a multiple alignment in which the induced alignment of any sequence with the center sequence is optimal. Therefore, as a heuristic, Start alignment chooses the center of the star to be the sequences s_i that maximizes

$$M_i = \sum_j OPT(s_i, s_j)$$

where OPT denotes the optimal score of aligning two sequences.

The running time of Start alignment is the same for the general tree alignment except that we have to account for finding $argmax_i M_i$. This can be done by performing all pairwise alignments and choosing the sequence s_i that maximizes M_i. Therefore, the running time of Star alignment is $O(k^2n^2 + k^2l)$.

Star alignment with the above heuristic has some nice properties in terms of approximating the optimal $SP-score$. The score of the Star alignment will be within a constant factor of the optimal $SP-score$, under some special pairwise schoring schemes.

References

Setubal J., Meidanis, J., Introduction to Molecular Biology, Chapter 3.

Gusfield D., Algorithms on Strings, Trees, and Sequences, Chapter 14.