DNA Alignment With Affine Gap Penalties

Laurel Schuster

Why Use Affine Gap Penalties?

When aligning two DNA sequences, one goal may be to infer the mutations that made them different. Though it's impossible to know the sequence of mutations for sure, the most likely explanation is the simplest, the one that minimizes the number of changes. For this class, we wrote a global alignment program that often outputs only one of multiple alignments with the highest possible score. One way to make a Needleman-Wunsch dynamic program more intelligent is to prioritize alignments with fewer longer insertions/deletions over those with many shorter insertions/deletions ('indels'). Adjacent gaps in the genetic code are more likely to represent a single indel than many, and as it stands, our program does not account for this at all. The program I've written handles this issue, and I will demonstrate the advantages with the following toy sequences:

String 1: ACCCCCCCCCA
String 2: ATCCTA

What changes could transform string 1 into string 2?

Scenario 1:

1. One T is inserted after the first A
2. Another T is inserted before the last A
3. Seven Cs are deleted

Scenario 2:

1. Two Cs become Ts
2. Five Cs are deleted

Scenario 3:

1. All Cs are deleted
2. TCCT is inserted

Suppose that a global alignment program is run according to the following scoring system:

- Match: +3
- Mismatch: -3
- Gap open (-σ): -1
- Gap extend (-ε): -1

```
def example():

  s1 = 'ACCCCCCCCCA'
  s2 = 'ATCCTA'

  score = [[3,-3,-3,-3],[-3,3,-3,-3],
           [-3,-3,3,-3],[-3,-3,-3,3]]
  σ = 1
  ε = 1

  alignment_score, align1, align2 = Align(s1, s2, score, σ, ε)
  print()
  print(align1)
  print(align2)
  print('score:', alignment_score)
```

This scoring mimics our original alignment scoring system, which does not differentiate between gap opening and gap extension.

```
A-CCCCCCCCC-A
ATC-------CTA
score: 3
```

This output matches scenario 1. It suggests that at least three evolutionary events have taken place. Note the score.

Now consider an only slightly different scoring system:

- Gap open (-σ): -2

```
def example():

  s1 = 'ACCCCCCCCCA'
  s2 = 'ATCCTA'

  score = [[3,-3,-3,-3],[-3,3,-3,-3],
           [-3,-3,3,-3],[-3,-3,-3,3]]
  σ = 2
  ε = 1

  alignment_score, align1, align2 = Align(s1, s2, score, σ, ε)
  print()
  print(align1)
  print(align2)
  print('score:', alignment_score)
```

This subtle change lowers the maximum score, and radically changes the optimal alignment.

```
ACCCCCCCCCA
A-----TCCTA
score: 0
```

This new output matches scenario 3. It reduces the minimum number of evolutionary events to two. The score is lower, but would be even more so if σ and ε were both raised to 2.

By penalizing gap opening more than gap extension, we arrive at a more likely alignment without lowering the score by too much.

How to Align Using Affine Gap Penalties:

- My master function is called **Align** and takes five parameters: two strings (*s1* and *s2*), a scoring matrix *score,* a gap open penalty *σ,* and a gap extend penalty *ε.*
- **Align** calls **dynamicProgram** on the same 5 parameters.
    - **initializeTable** builds six matrices of zeros: *middle, lower, upper, M_backtrack, L_backtrack,* and *U_backtrack*. Each table has (len(*s1*) + 1) rows and (len(*s2*) + 1) columns.
    - **populateScore** uses the scoring matrix to fill the middle table with the incoming 'edge' weights for any diagonal moves.
        - **scoreLookup** uses *score* to determine the appropriate weight for each 'edge' based on the positions of A, C, G, and T
    - *Middle, lower,* and *upper* form a list *parkingStructure* and their respective backtrack tables form a list *backtrackStructure.*
    - A triply nested loop executes the dynamic population of the scores as it moves through the structure. With each step, a list of possible scores *candidates* is generated by a recurrence relation, and another list *directions* holds the corresponding source (initials of table and direction) of the imagined entering edge. The recurrence relation used to build *candidates* depends on which level (*k*) of *parkingStructure* is being filled. The maximum candidate score becomes the value of the 'node' in matrix *k* at row *i,* column *j.*

```
for i in range(len(middle)):
    for j in range(len(middle[0])):
        for k in range(len(parkingStructure)):
```

RECURRENCE RELATION

- middle[i][j] = max{
  - If i > 0 and j > 0:

    Diagonal movement within middle table or from any other table, adding initial middle score to running total.

    - middle[i-1][j-1] + middle[i][j]
      - Middle Diagonal ('MD')
    - if i > 1:

      lower[i-1][j-1] - 0 + middle[i][j]
    - Lower Diagonal ('LD')
    - if j > 1:

      upper[i-1][j-1] - 0 + middle[i][j]
      - Upper Diagonal ('UD')
  - Elif i > 0:

    Though it usually works to populate the middle table and then the others, the first column causes problems in representing southward movement, so will be based off of values in middle table to make things easier.

    - If i == 1:

      -σ
      - Lower to Middle ('LM')
    - If i > 1:

      middle[i-1][j] - ε
      - Lower to Middle ('LM')
- lower[i][j] = max{

  Southward movement within lower table or from middle table.

  - If i > 1:

    Gap extension only possible after the section of the table is reached where a gap has definitely been opened.

    lower[i-1][j] - d
    - Lower South ('LS')
  - If (i > 0 and j != 0) or (i == 1 and j == 0):

    middle[i-1][j] - σ

        - Middle South ('MS')
    - upper[i][j] = max{

      Eastward movement within upper table or from middle table.

        - If j > 1:

          As with the lower table, only open gaps can be extended.

          upper[i][j-1] - ε
            - Upper East ('UE')
        - If (j > 0 and i != 0) or (i ==0 and j == 1):

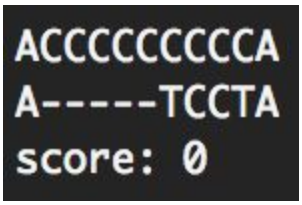
          middle[i][j-1] - σ
            - Middle East ('ME')
    - If none of the conditions have been met, the candidates list is empty, so the node is scored a 0 and the backtrack value is '*'
    - After assigning a score to (*i, j*) in every table, (*i, j*) is reevaluated, checking whether a gap should be closed in one aligned string and opened in the other. That is…

```
if k == 2:
  if 'ME' in L_backtrack:
    if lower[i][j] - σ > upper[i][j]:
      upper[i][j] = lower[i][j] - σ
      U_backtrack[i][j] = 'LU'
  if 'MS' in U_backtrack:
    if lower[i][j] - σ > upper[i][j]:
      lower[i][j] = upper[i][j] - σ
      L_backtrack[i][j] = 'UL'
```

  - **dynamicProgram** prints the six matrices, returns the six matrices, and returns the maximum *i* and *j* values (*maxi, maxj*), which will be used to backtrack.
- **Align** calls **backtrack** on *s1, s2,* and the outputs of **dynamicProgram.**
  - **chooseBest** selects the level of *parkingStructure* with the highest final score and assigns it to *backtrack. backtrack*[*maxi*][*maxj*] is assigned to *alignment_score*
  - While *i* > 0 or *j* > 0, a series of if statements builds two strings representing the alignment, *align1* and *align2*
    - If the direction was diagonal ('D'), the location has a match or a mismatch
      - the *i* - 1 character of *s1* is prepended to *align1*
      - the *j* -1 character of *s2* is prepended to *align2*
      - *i* and *j* are decremented

        - If the direction was south ('S'), the location has a deletion from *s1*
            - the *i* - 1 character of *s1* is prepended to *align1*
            - '-' is prepended to *align2*
            - *i* is decremented
        - If the direction was east ('E'), the location has an insertion to *s2*
            - '-' is prepended to *align1*
            - the *j* - 1 character of *s2* is prepended to *align2*
            - *j* is decremented
        - *backtrack* is reassigned to the source table ('M,' 'L,' or 'U'). Note that if the direction (second letter) is M, L, or U, nothing is added to the alignment strings, and all that changes is the backtrack table from which the next set of directions will be pulled for the same *i, j*
    - **backtrack** returns *align1, align2,* and *alignment_score*
- **Align** returns the outputs of **backtrack**

Possible Applications/Extensions:



Let's return to the toy sequences we played with earlier. These are very short strings with very low similarity, but what if they were bookended by larger sections of high similarity? I wouldn't want the global alignment to suffer because of so many indels created by maximizing the matches; I would rather shift the entire scoring system a little so that I could see where the regions of low similarity are without misrepresenting the evolutionary history.

Scenario 3—one large deletion and one large insertion—is the most likely. Improvements could be made to this program. In this example, the insertion TCCT is scored as two matches and two mismatches, whereas by evolutionary scenario 3, this should not be awarded points for the matching Cs, since they represent mutation even though they match. I believe this could be handled by a similar algorithm, though it would be tricky to decide how many matching bases should count as part of larger blocks of inserted mismatch before the mismatches should be read as transformations instead.

Ideal sequence alignment walks a fine line between too many gaps and not enough of them, allowing the most bases possible to match without rendering the alignment meaningless. At the end of the day, there is no universal gap penalty that will output the most likely alignment for every pair of sequences. Evolutionary distance, purpose of analysis, and degree of variation in similarity within an alignment should be considered when selecting gap penalties. This program allows for a higher degree of control and specificity. High penalties for gap opening result in alignments that are easy to read. At a glance, one can differentiate blocks of the alignment with high similarity from those with low similarity. This isolation of sections for analysis can help when identifying the genetic basis for small structural differences between proteins, and might be another way to identify synteny blocks. The inclusion of multiple types of gap penalties allows for tinkering with alignments until they are the most useful they can be.