Gene Sequencing Report

Niels Turley

1. Source code

#!/usr/bin/python3  
import math  
  
from cell import Cell  
  
from which\_pyqt import PYQT\_VER  
if PYQT\_VER == 'PYQT5':  
 from PyQt5.QtCore import QLineF, QPointF  
elif PYQT\_VER == 'PYQT4':  
 from PyQt4.QtCore import QLineF, QPointF  
elif PYQT\_VER == 'PYQT6':  
 from PyQt6.QtCore import QLineF, QPointF  
else:  
 raise Exception('Unsupported Version of PyQt: {}'.format(PYQT\_VER))  
  
import random  
  
# Used to compute the bandwidth for banded version  
MAXINDELS = 3  
  
# Used to implement Needleman-Wunsch scoring  
MATCH = -3  
INDEL = 5  
SUB = 1  
  
class GeneSequencing:  
  
 def \_\_init\_\_( self ):  
 pass  
  
 # This is the method called by the GUI. \_seq1\_ and \_seq2\_ are two sequences to be aligned, \_banded\_ is a boolean that tells  
 # you whether you should compute a banded alignment or full alignment, and \_align\_length\_ tells you  
 # how many base pairs to use in computing the alignment  
  
 def align( self, seq1, seq2, banded, align\_length):  
 self.banded = banded  
 self.MaxCharactersToAlign = align\_length  
 alignment1 = ''  
 alignment2 = ''  
 score = 0  
  
 # If banded, compute the bandwidth  
 if banded:  
 score, alignment1, alignment2 = self.compute\_banded\_alignment(seq1, seq2)  
 if score == math.inf:  
 return{'align\_cost':score, 'seqi\_first100':alignment1, 'seqj\_first100':alignment2}  
 else:  
 score, alignment1, alignment2 = self.compute\_alignment(seq1, seq2)  
  
 alignment1 = alignment1[:100]  
 alignment2 = alignment2[:100]  
  
 return {'align\_cost':score, 'seqi\_first100':alignment1, 'seqj\_first100':alignment2}  
  
 # Time complexity: O(n \* m). n and m are the lengths of the sequences. We have to iterate through each cell once,  
 # computing the best path/score each time.  
   
 # Space complexity: O(n \* m). n and m are the lengths of the sequences.  
 # The matrix contains a cell for each possible edit distance.  
 def compute\_alignment(self, seq1, seq2):  
 length\_1 = min(len(seq1) + 1, self.MaxCharactersToAlign + 1)  
 length\_2 = min(len(seq2) + 1, self.MaxCharactersToAlign + 1)  
 # Create a matrix of cells  
 matrix = [[Cell(0, None, '', '') for i in range(length\_2)] for j in range(length\_1)]  
  
 # Initialize the first row and column  
 for i in range(length\_1):  
 matrix[i][0] = Cell(i \* INDEL, (i - 1, 0), seq1[i - 1], '-')  
 for j in range(length\_2):  
 matrix[0][j] = Cell(j \* INDEL, (0, j - 1), '-', seq2[j - 1])  
  
 # Fill in the rest of the matrix  
 for i in range(1, length\_1):  
 for j in range(1, length\_2):  
 # Calculate the score for a match/mismatch  
 if seq1[i - 1] == seq2[j - 1]:  
 match\_score = MATCH  
 else:  
 match\_score = SUB  
 match = matrix[i - 1][j - 1].score + match\_score  
  
  
 # Calculate the score for an insertion  
 insert = matrix[i - 1][j].score + INDEL  
  
 # Calculate the score for a deletion  
 delete = matrix[i][j - 1].score + INDEL  
  
 # Determine the best score. tie goes insert first, then delete, then match  
 if insert <= match and insert <= delete:  
 matrix[i][j] = Cell(insert, (i - 1, j), seq1[i - 1], '-')  
 elif delete <= match and delete <= insert:  
 matrix[i][j] = Cell(delete, (i, j - 1), '-', seq2[j - 1])  
 else:  
 matrix[i][j] = Cell(match, (i - 1, j - 1), seq1[i - 1], seq2[j - 1])  
  
 # Traceback to find the alignment  
 alignment1 = ''  
 alignment2 = ''  
 i = length\_1 - 1  
 j = length\_2 - 1  
 while i > 0 or j > 0:  
 alignment1 = matrix[i][j].char1 + alignment1  
 alignment2 = matrix[i][j].char2 + alignment2  
 i, j = matrix[i][j].parent  
  
 return matrix[length\_1 - 1][length\_2 - 1].score, alignment1, alignment2  
  
 # Time complexity: O(n \* k). n is the length of the first sequence, and k is (MAXINDELS \* 2 + 1). We iterate   
 # through the larger n \* m matrix, but only in a narrow, diagonal band of cells, where each cell is only   
 # MAXINDELS (in this case, 3 indels) apart from any other cell. Each cell is visited once to calculate the best   
 # score/path.   
   
 # Space complexity: O(n \* k). n is the length of the first sequence, and k is (MAXINDELS \* 2 + 1). We only create  
 # and store a matrix for the band of cells.  
 def compute\_banded\_alignment(self, seq1, seq2):  
 length\_1 = min(len(seq1) + 1, self.MaxCharactersToAlign + 1)  
 length\_2 = min(len(seq2) + 1, self.MaxCharactersToAlign + 1)  
 if abs(length\_1 - length\_2) > MAXINDELS:  
 return math.inf, 'No Alignment Possible', 'No Alignment Possible'  
  
 # Create a matrix of cells  
 matrix = [[Cell(math.inf, None, '', '') for i in range(MAXINDELS \* 2 + 1)] for j in range(length\_1)]  
  
 # Initialize the first row and column  
 for i in range(MAXINDELS + 1):  
 matrix[i][0] = Cell(i \* INDEL, (i - 1, 0), seq1[i - 1], '-')  
 for j in range(MAXINDELS + 1):  
 matrix[0][j] = Cell(j \* INDEL, (0, j - 1), '-', seq2[j - 1])  
  
 # Fill in the rest of the matrix  
 for i in range(1, length\_1):  
 for j in range(MAXINDELS \* 2 + 1):  
 start\_or\_end = False  
 if i - MAXINDELS <= 0:  
 k = j  
 start\_or\_end = True  
 if k > i + MAXINDELS or j == 0:  
 continue  
 elif length\_1 - MAXINDELS <= i:  
 k = length\_1 - (2 \* MAXINDELS) + j - 1  
 start\_or\_end = True  
 if k < i - MAXINDELS:  
 continue  
 else:  
 k = i - MAXINDELS + j  
  
 # use different parent cell scheme for start\_or\_end  
 if start\_or\_end:  
 # Calculate the score for a match/mismatch  
 if seq1[i - 1] == seq2[k - 1]:  
 match\_score = MATCH  
 else:  
 match\_score = SUB  
 match = matrix[i - 1][j - 1].score + match\_score  
  
 # Calculate the score for an insertion  
 insert = matrix[i - 1][j].score + INDEL  
  
 # Calculate the score for a deletion  
 delete = matrix[i][j - 1].score + INDEL  
  
 # Determine the best score. tie goes insert first, then delete, then match  
 if insert <= match and insert <= delete:  
 matrix[i][j] = Cell(insert, (i - 1, j), seq1[i - 1], '-')  
 elif delete <= match and delete <= insert:  
 matrix[i][j] = Cell(delete, (i, j - 1), '-', seq2[k - 1])  
 else:  
 matrix[i][j] = Cell(match, (i - 1, j - 1), seq1[i - 1], seq2[k - 1])  
 else:  
 # Calculate the score for a match/mismatch  
 if seq1[i - 1] == seq2[k - 1]:  
 match\_score = MATCH  
 else:  
 match\_score = SUB  
 match = matrix[i - 1][j].score + match\_score  
  
 # Calculate the score for an insertion  
 if j == MAXINDELS \* 2:  
 insert = math.inf  
 else:  
 insert = matrix[i - 1][j + 1].score + INDEL  
  
 # Calculate the score for a deletion  
 delete = matrix[i][j - 1].score + INDEL  
  
 # Determine the best score. tie goes insert first, then delete, then match  
 if insert <= match and insert <= delete:  
 matrix[i][j] = Cell(insert, (i - 1, j + 1), seq1[i - 1], '-')  
 elif delete <= match and delete <= insert:  
 matrix[i][j] = Cell(delete, (i, j - 1), '-', seq2[k - 1])  
 else:  
 matrix[i][j] = Cell(match, (i - 1, j), seq1[i - 1], seq2[k - 1])  
  
 # Traceback to find the alignment  
 alignment1 = ''  
 alignment2 = ''  
 i = length\_1 - 1  
 j = MAXINDELS \* 2  
 while i > 0 or j > 0:  
 alignment1 = matrix[i][j].char1 + alignment1  
 alignment2 = matrix[i][j].char2 + alignment2  
 i, j = matrix[i][j].parent  
  
 return matrix[length\_1 - 1][MAXINDELS \* 2].score, alignment1, alignment2

class Cell:  
 def \_\_init\_\_(self, score, parent, char1, char2):  
 self.score = score  
 self.parent = parent  
 self.char1 = char1  
 self.char2 = char2

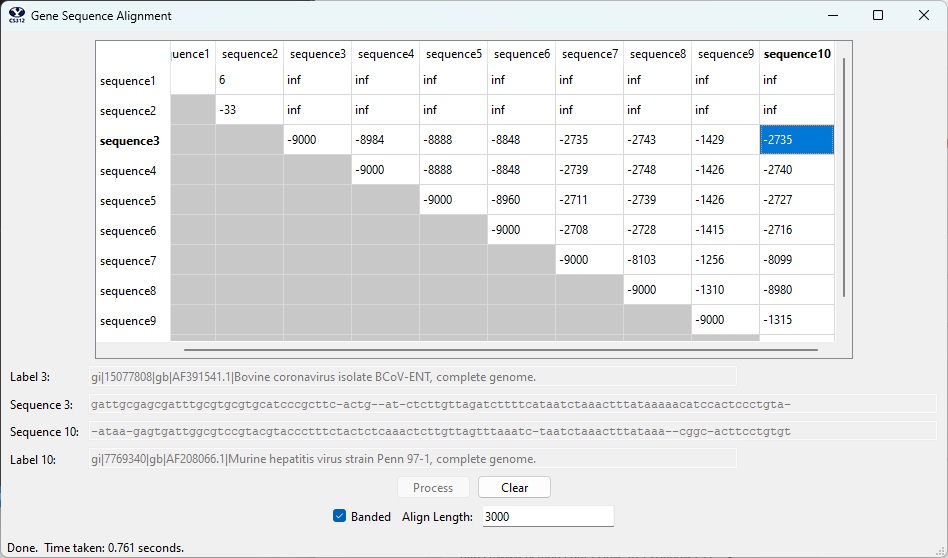
1. Time and space complexity discussion
   1. compute\_alignment()
      1. Time: O(n \* m), where n is the length of sequence 1 and m is the length of sequence 2. We have to iterate through each cell in the matrix, calculating the best adjacent path and score (linear time) for each cell. There are additional calculations, like the traceback from the last cell to the first following the best path and the initial base cases, but those are all superseded by the n \* m.
      2. Space: O(n \* m). The matrix stores a Cell object in each cell, which only contains a couple variables for the score, parent cell, and the characters. So the total space is just the n \* m array.
   2. compute\_banded\_alignment()
      1. Time: O(n \* k), where n is the length of sequence 1 and k is the MAXINDELS \* 2 + 1. We have to iterate through a narrow, diagonal band down the edit distance matrix—only through the cells that are within three MAXINDELS from the diagonal. We calculate the best adjacent path and score (linear time) for each cell. There are additional calculations, like the traceback from the last cell to the first following the best path and the initial base cases, but those are all superseded by the n \* k.
      2. Space: O(n \* k). The matrix stores a Cell object in each cell, which only contains a couple variables for the score, parent cell, and the characters. So the total space is just the n \* k array.
2. Screenshots + alignment for r3, c10

Graphical user interface

Description automatically generatedUnbanded, n = 1000

**Seq3: gattgcgagcgatttgcgtgcgtgcatcccgcttc-actg--at-ctcttgttagatcttttcataatctaaactttataaaaacatccactccctgta-**

**Seq10: -ataa-gagtgattggcgtccgtacgtaccctttctactctcaaactcttgttagtttaaatc-taatctaaactttataaa--cggc-acttcctgtgt**

Banded, n = 3000

**Seq3: gattgcgagcgatttgcgtgcgtgcatcccgcttc-actg--at-ctcttgttagatcttttcataatctaaactttataaaaacatccactccctgta-**

**Seq10: -ataa-gagtgattggcgtccgtacgtaccctttctactctcaaactcttgttagtttaaatc-taatctaaactttataaa--cggc-acttcctgtgt**