Trevor Ashby

Gene Sequencing

|  |  |
| --- | --- |
| 1. Time and Space Complexity | |
| Unrestricted | Banded |
| Space & Time O(n\*m)  initialize array **->O(n\*m)**  for I in len(seq1) **->O(n)**  set axis  for I in len(seq2) **->O(m)**  set axis  for I in len(seq1) **-> (O(n\*m))**  for j in len(seq2)  set array[j][i] = min(left cell, top cell, diag cell)  go through back pointers **->O(n)**  build alignment string based on pointers  reverse alignments **-> O(n) O(m)** | Space & Time: O(n\*k)  Initialize array w/ shortest seq **->O(n\*k)**  For I in len(k) **->O(k)**  Set axis  For I in len(n) **->O(n)**  Set axis  For I in len(n) **->O(n\*k)**  For j in len(k)  If I or j out of range, set to inf  Set array[n][k] = min(left cell, top cell, diag cell)  Go through back pointers **->O(n)**  Build alignment string based on pointers  Reverse alignments **-> O(n) O(m)** |
|  |  |

|  |
| --- |
| 1. Alignment Extraction |
| To extract the alignment, I inserted tuples into my matrix that had the edit distance value, as well as a character that determined what kind of transition the value was taken from. For example, ‘s’ represented that the characters in the strings were the same and that the value that was paired with it was obtained by subtracting 3 from the diagonal (Non-banded) or above (banded) value. Later, I back tracked from the optimal value, following the trail of character pointers that were left, and depending on the character, I would append either a character from the strings, or a ‘-‘ depending on the value of the pointer. |

|  |
| --- |
| 1. Results |
|  |
|  |

|  |
| --- |
| 1. Results 2: Extracted Alignments |
| Un-Banded    Banded |

|  |
| --- |
| 1. Code |
| #!/usr/bin/python3  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 else:  raise Exception('Unsupported Version of PyQt: {}'.format(PYQT\_VER))  import math import time 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):  t0 = time.clock()  self.banded = banded  self.MaxCharactersToAlign = align\_length   # First, determine if the full length of the string can be used, or if it has to be  # shortened based on the align\_length parameter  if len(seq1) > align\_length:  seq1 = seq1[:align\_length]  if len(seq2) > align\_length:  seq2 = seq2[:align\_length]   cnt = 0  # if a banded check is to be ran  ##########################  # Time Complexity: O(k\*n)  # Space Complexity: O(k\*n)  ##########################  if (self.banded):  bandwidth = 7  distance = 3  width = 0  seq1Used = True   # if the strings that are being compared are too different in length, return not possible  if (abs(len(seq1) - len(seq2)) > bandwidth):  return {'align\_cost': float('inf'), 'seqi\_first100': "No Alignment Possible", 'seqj\_first100': "No Alignment Possible"}  myArray = []   # initialize array based on shortest sequence  if len(seq1) < len(seq2):  myArray = [[(-1, 'n') for x in range(bandwidth)] for y in range(len(seq1)+1)]  width = len(seq1)+1  seq1Used = True  elif len(seq2) < len(seq1):  myArray = [[(-1, 'n') for x in range(bandwidth)] for y in range(len(seq2)+1)]  width = len(seq2)+1  seq1Used = False  else:  myArray = [[(-1, 'n') for x in range(bandwidth)] for y in range(len(seq1)+1)]  width = len(seq1)+1  seq1Used = True   #initialize top row of array with axis values  temp = 0  for col in range(bandwidth):  if col < 3:  myArray[0][col] = (float('inf'), 'x')  else:  myArray[0][col] = (temp, 'b')  temp += 5   # initialize diagonal of array with axis values  temp = 15  row = 3  for col in range(distance):  myArray[row][col] = (temp, 'b')  temp -= 5  row -= 1   # initialize 1,1  myArray[1][1] = (float('inf'), 'x')   # initialize 1,0  myArray[1][0] = (float('inf'), 'x')   # initialize 2,0  myArray[2][0] = (float('inf'), 'x')   distance = 3  end = bandwidth  begin = distance   # run through each value in the matrix starting at 0,0  for row in range(0, width):  for col in range(0, bandwidth):   # if the row and col fits within the region available to edit, change the value  if (row + col - 3 - (abs(len(seq1) - len(seq2)))) >= width:  myArray[row][col] = (float('inf'), 'x')   # if the value at the desired location is infinity or set to 'b' for base, do not edit matrix cell  elif myArray[row][col][0] == float('inf') or myArray[row][col][1] == 'b':  continue  else:   wereSame = False  optimalSame = (myArray[row-1][col][0] + MATCH, 's')   # if seq1 is the shorter sequence, use the following for string comparison  if seq1Used:  adj = col - 3  if (seq1[row-1] == seq2[row+adj-1]):  wereSame = True  # if not, use this instead  else:  adj = col - 3  if (seq1[row+adj-1] == seq2[row-1]):  wereSame = True   # check left cell  if col-1 < 0:  optimalLeft = (float('inf'), 'x')  else:  optimalLeft = (myArray[row][col-1][0] + INDEL, 'l')   # check top cell  if col+1 >= bandwidth:  optimalTop = (float('inf'), 'x')  else:  optimalTop = (myArray[row-1][col+1][0] + INDEL, 'd')   # check diag cell  if row-1 < 0:  optimalDiag = (float('inf'), 'x')  else:  optimalDiag = (myArray[row-1][col][0] + SUB, 't')   # if the values are the same, consider the optimalSame value, if not, ignore  if wereSame:  lowest = min(optimalSame[0], optimalTop[0], optimalDiag[0], optimalLeft[0])  else:  lowest = min(optimalTop[0], optimalDiag[0], optimalLeft[0])   # if same is lowest  if lowest == optimalSame[0] and wereSame:  myArray[row][col] = optimalSame  # if left is lowest  if lowest == optimalLeft[0]:  myArray[row][col] = optimalLeft  # if diag is lowest  elif lowest == optimalTop[0]:  myArray[row][col] = optimalTop  # if top is lowest  elif lowest == optimalDiag[0]:  myArray[row][col] = optimalDiag  end -= 1  if begin > 0:  begin -= 1  # iterate through the last row until the value of infinity is hit. Set the previous value before that  # to the optimal score  start\_row, start\_col = 0, 0  for col in range(bandwidth):  if myArray[width-1][col][0] == float('inf'):  score = myArray[width-1][col-1][0]  start\_row = width-1  start\_col = col-1  break   alignment1 = ""  alignment2 = ""  # loop through the previous pointer paths from the optimal value  # until a 'b' base value is hit  while(1):  tuple = myArray[start\_row][start\_col]  adj = start\_col - 3  if tuple[1] == 'b':  #alignment1 += seq1[start\_row-1]  #alignment2 += seq2[0]  break  if tuple[1] == 's':  alignment1 += seq1[start\_row-1]  alignment2 += seq2[start\_row+adj-1]  start\_row -= 1  elif tuple[1] == 'l':  alignment1 += '-'  alignment2 += seq2[start\_row+adj-1] # may need minus 1 here  start\_col -= 1  elif tuple[1] == 't':  alignment1 += seq1[start\_row-1]  alignment2 += seq2[start\_row+adj-1] # may need minus 1 here  start\_row -= 1   elif tuple[1] == 'd':  alignment1 += seq1[start\_row-1]  alignment2 += '-'  start\_row -= 1  start\_col += 1  # if un-banded calculation is desired  ##########################  # Time Complexity: O(n\*m)  # Space Complexity: O(n\*m)  ##########################  else:  # initialize array  myArray = [[(-1, 'b') for x in range(len(seq1) + 1)] for y in range(len(seq2) + 1)]   # initialize top row with axis  temp = 0  for i in range(len(seq1)+1):  myArray[0][i] = (temp, 'b')  temp += 5   # initialize left col with axis  temp = 5  for i in range(1, len(seq2)+1):  myArray[i][0] = (temp, 'b')  temp += 5   # begin Needleman/Wunsch  for i in range(1, len(seq1)+1):  for j in range(1, len(seq2)+1):   # check if the same character  wereSame = False  optimalSame = (myArray[j - 1][i - 1][0] + MATCH, 's')  if (seq1[i - 1] == seq2[j - 1]):  wereSame = True   # check top cell  optimalTop = (myArray[j - 1][i][0] + INDEL, 't')   # check top left cell  optimalDiag = (myArray[j - 1][i - 1][0] + SUB, 'd')   # check left cell  optimalLeft = (myArray[j][i - 1][0] + INDEL, 'l')   # if the characters in the strings are the same, consider the optimalSame value. If not, igore it  if wereSame:  lowest = min(optimalSame[0], optimalTop[0], optimalDiag[0], optimalLeft[0])  else:  lowest = min(optimalTop[0], optimalDiag[0], optimalLeft[0])   # if same is lowest  if lowest == optimalSame[0] and wereSame:  myArray[j][i] = optimalSame  # if left is lowest  if lowest == optimalLeft[0]:  myArray[j][i] = optimalLeft  # if top is lowest  elif lowest == optimalTop[0]:  myArray[j][i] = optimalTop  # if diag is lowest  elif lowest == optimalDiag[0]:  myArray[j][i] = optimalDiag   score = myArray[len(seq2)][len(seq1)][0]  alignment1 = ""  alignment2 = ""  row = len(seq2)  col = len(seq1)  # loop through the previous pointer paths from the optimal value  # until row and col = 0  while(row > 0 and col > 0):  tuple = myArray[row][col]  if tuple[1] == 'b':  alignment1 += seq1[col-1]  alignment2 += seq2[row-1]  if tuple[1] == 's':  alignment1 += seq1[col-1]  alignment2 += seq2[row-1]  row -= 1  col -= 1  elif tuple[1] == 'l':  alignment1 += seq1[col-1]  alignment2 += '-'  col -= 1  elif tuple[1] == 't':  alignment1 += '-'  alignment2 += seq2[row-1]  row -= 1  elif tuple[1] == 'd':  alignment1 += seq1[col-1]  alignment2 += seq2[row-1]  row -= 1  col -= 1  alignment1 = alignment1[::-1]  alignment2 = alignment2[::-1]   t1 = time.clock() - t0  return {'align\_cost':score, 'seqi\_first100':alignment1[0:100], 'seqj\_first100':alignment2[0:100]} |